1995

Scalability study in parallel computing

Mark Alan Fienup
Iowa State University

Scalability study in parallel computing

by

Mark Alan Fienup

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Computer Science
Major: Computer Science

Approved:
Signature was redacted for privacy.
In Charge of Major Work
Signature was redacted for privacy.
For the Major Department
Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa

1995

To my wife, Virginia, and daughter, Ann.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

1.1. Motivation

One of the major goals of parallel computing is to decrease the execution-time of a computing task. For sequential programs, there are often several algorithms for solving a task, but usually a simple time-complexity analysis using "big-oh" notation suffices in determining the better algorithm. When trying to solve a task on a parallel computer, several complicating factors arise: (1) "How should the problem be decomposed on the processors?", (2) "How does the communication network's topology impact performance?", (3) "If I run my problem using more processors, how much improvement can I expect?", (4) "Given a choice of parallel computers to run my task, which should I choose?", and (5) "If I want to solve a bigger problem, using more processors what kind of performance can I expect?".

To help answer many of these questions several scalability metrics have been proposed. These metrics differ in the scaling assumptions that they make. For example, Amdahl's law [2] assumes that the problem size is fixed as the number of processors is increased, the isoefficiency function [8] keeps the parallel computers efficiency fixed by allowing the problem size to grow with the number of processors, and in [23] scaling to control the simulation error in scientific applications is discussed. In this thesis, a new scalablity metric, called CMP-scalability *(Constant-Memory-per-Processor) [5] [6], is proposed that assumes that the memory used per processor is constant as the problem size grows.

As you might expect, the CMP scalability metric is tailored to answering question (5) above: "If I want to solve a bigger problem, using more processors what kind of performance can I expect?". Specifically, the CMP-scalability function describes the asymptotic rate of grow of the speedup function under the constant-memory-per-processor scaling assumption. This seems like a natural question to ask as parallel-computer users often want to run larger problems. Since the memory available on each processor is fixed, more processors are required to store the larger problem and execute the problem in a shorter amount of time. Thus, CMP-scalability seems to be based on a reasonable scaling assumption.

Parallel implementations of Cannon's matrix multiplication (MM) [4], Gauss-Jordan Elimination with partial pivoting (GJE), and Faster Fourier Transform (FFT) [6] on the MasPar architecture (a SIMD machine with a two-dimensional array of processors) were performed to evaluate the CMP-scalability metric. These algorithms were chosen because of their widely varying computation to communication ratios. The CMP-scalability metric predicted that the speedup of matrix multiplication would grow linearly, $O(P)$, with the number of processors, the speedup of Gauss-Jordan Elimination would grow as $O(\sqrt{P})$ with the number of processors, and the speedup of Fast Fourier Transform would grow as $O(\sqrt{P}\log_2 P)$ with the number of processors. It was somewhat surprising that the communication intensive Fast Fourier Transform was predicted to outperform the computationally more intensive Gauss-Jordan Elimination.

Experimental studies on a 16K-processor MasPar MP-1 and 4K-processor MP-2 did

not seem to behave as predicted by the CMP-scalability metric. The speedup plots, under constant-memory-per-processor scaling, for all three algorithms appeared to be roughly linear with the slopes of 0.89, 0.76, and 0.62 for matrix multiplication, Gauss-Jordan Elimination, and Fast Fourier Transform, respectively. At this point, the formulas describing the execution time of each algorithm were scrutinized, refined, and experimentally verified since these formulas were used in the CMP-scalability analysis. While some small errors in the formulas were discovered, the CMP-scalability analysis was not affected.

To help explain the experimental results, other scalability metrics, especially the isoefficiency function, were used to analyze the chosen algorithms. The isoefficiency analysis of the three algorithms on a mesh architecture predicted that matrix multiplication would scale better than Gauss-Jordan Elimination which would scale much better than Fast Fourier Transform. Surprisingly, the predicted relative order of the scalability for the three algorithms was different for the isoefficiency metric than the CMP-scalability metric. While the relative scalability predicted by the isoefficiency function matched the experimental study, the magnitude of difference between the Gauss-Jordan Elimination and Fast Fourier Transform was not experimentally observed.

1.2. Questions to be Answered in the Thesis

The following questions will be answered by the thesis:

1) How can the CMP and isoefficiency-scalability metrics predict different relative scalability for the Gauss-Jordan Elimination and Fast Fourier Transform algorithms?

2) Why don't the experimental results on the MasPar computers agree with the CMP and

isoefficiency-scalability analyses?

3) Would the predicted scalability analysis be observed on a different parallel computer with different machine parameters?

1.3. Organization of Thesis and Summary of Results

The thesis starts in Chapter 2 with a review of the relevant literature to provide necessary background information. Scaling alternative and other scalability metrics are reviewed. In light of the questions to be answered by the thesis, particular emphasis will be placed on constant-memory-per-processor scaling and the isoefficiency-scalability metric.

Chapter 3 discusses the parallel algorithms for Cannon's matrix multiplication, Gauss-Jordan Elimination, and Fast Fourier Transform on a two-dimensional mesh and on a hypercube parallel computer. For each algorithm, execution-time formulas are developed.

These formulas are used in Chapter 4 to demonstrate how the asymptotic CMP and isoefficiency-scalability metrics can be calculated with a minimum amount of work. The apparently conflicting scalability results between the CMP and isoefficiency-scalability metrics for Gauss-Jordan Elimination and Fast Fourier Transform on a mesh are identified.

At this point question number (1) is answered by observing that for these parallel algorithms performance is really a surface over the plane of two independent variables: the number of processors and the problem size. The different scaling assumptions of the CMP and isoefficiency-scalability metrics describe different planar cross-sections in this three-dimensional space with the CMP and isoefficiency functions being the intersections of these planes with the performance surface. Thus, the CMP and isoefficiency functions

actually provide complimentary information about the performance surfaces and not conflicting information. Using this complimentary information from both the CMP and isoefficiency functions, two theorems are proven that predict the relative change in performance if only the number of processors is varied, or if only the size of the problem is varied. Chapter 4 concludes with an examination of why most algorithms are not fixed-time scalable.

The general algorithms described in Chapter 3 for a mesh architecture must be refined to provide good performance on the MasPar architecture. Chapter 5 describes the MasPar specific "tricks" used when implementing these algorithms. Modified computation, communication, and memory-access execution-time formulas for the MasPar implementations are derived. Further refinements that include miscellaneous overhead and memory overlapping are made to these formulas in Chapter 6. These refinements are based on experimental timings on the MasPar computers. Verification of the accuracy of the resulting execution-time formulas is also provided in this chapter.

These detailed execution-time formulas for the algorithms are used in Chapter 7 to answer the remaining two questions: (2) "Why don't the experimental results on the MasPar computers agree with the CMP and isoefficiency-scalability analyses?" and (3) "Would the predicted scalability analysis be observed on a different parallel computer with different machine parameters?".

The answer to question (2) is that the CMP and isoefficiency-scalability metrics are asymptotic scalability metrics, i.e., they are only guaranteed to be true for a sufficiently large

number of processors. Even though the MasPar MP-1 has 16K processors, the machine specific parameters are such that the asymptotic behavior is not observed. Chapter 7 contains a detailed analysis of each algorithm's constants to predict the inaccuracies of the CMP and isoefficiency's predictions for a varying numbers of processors.

Question (3) is answered by modifying the execution-time formulas to speedup the computation or communication over the MP-1 machine parameters. Computation and communication speedups of ten, fifty, and one-hundred are examined for their effects on the accuracy of the CMP and isoefficiency-scalability metrics. It was found that small improvements in the computation speed dramatically improved the accuracy of the CMP-scalability predictions. The accuracy of the isoefficiency-scalability metric was found to be more algorithm dependent than the CMP-scalability metric when varying the machine parameters.

Chapter 8 concludes the thesis by highlighting the important results and discussing further areas of research.

CHAPTER 2. LITERATURE REVIEW

2.1. Terminology

Let P represent the number of processors, and N represent the problem size in terms of memory usage, such as number of elements, bytes, etc. The *speedup* of a parallel algorithm is traditionally defined as

$$Speedup(P, N) = \frac{T_1(N)}{T_P(N)}, \tag{2.1}$$

where $T_1(N)$ is defined to be the execution-time for the best sequential algorithm on a problem of size N, and $T_P(N)$ is defined to be the execution-time of the parallel algorithm using P processors. Speedup represents the reduction of execution time over the sequential algorithm. Often $T_1(N)$ is approximated in the literature by the parallel algorithm run on one processor, which tends to artificially boost the reported speedup [3][25]. Closely related to speedup is the notion of *efficiency* which is

$$Efficiency(P, N) = \frac{Speedup(P,N)}{P} = \frac{T_1(N)}{P*T_P(N)}, \tag{2.2}$$

which represents the utilization of the processors in the parallel computer.

2.2. Notions of Scaling

Amdahl [2] showed that parallel speedup is bounded if the problem size in fixed as the number of processors is increased. Define *fixed-size* scaling to be when the number of processors is scaled on a fixed-size problem. Specifically, Amdahl's law [2] says

$$\text{Speedup}(P, N) \leq \frac{1}{s + (1-s)/P}, \tag{2.3}$$

where s is the fraction of the sequential execution time that cannot be parallelized. Unfortunately, even for small values of s the speedup is severally restricted.

Amdahl's law implies that some convention must be adopted for scaling the problem size with the number of processors [24]. Gustafson, Montry, and Benner [13] demonstrated that Amdahl's law does not directly apply to *scaled speedup* where the problem is allowed to grow as the number of processors increased. Scaled speedup is also called *memory-bounded, memory-constrained,* or *constant-memory-per-processor (CMP) scaling.* Gustafson [15] has extended the traditional definition of speedup to include *memory-bounded speedup* which is defined as

$$\text{Memory-bounded Speedup}(P, N^*) = \frac{T_1(N^*)}{T_P(N^*)}, \tag{2.4}$$

where the sequential and parallel execution times are measured on the scaled problem size, N^*. The CMP-scalability metric presented later in this chapter is based on this notion of memory-bounded speedup.

Another scaling notion, called *fixed-time(/time-constrained) scaling,* [15] uses a fixed amount of time to solve as large of a problem as possible. Letting N' denote the largest problem that can be run in the fixed amount of time, the *fixed-time speedup* is defined as

$$\text{Fixed-time Speedup}(P, N') = \frac{T_1(N')}{T_P(N')}. \tag{2.5}$$

Gustafson's Slalom benchmark program [15] uses fixed-time scaling to measure the performance of a wide range of computers. In Chapter 4 of this thesis, it is shown that for a large class of algorithms N' is bounded even if the number of processors is unlimited.

In *constant efficiency scaling* the problem size is allowed to grow sufficiently fast as the number of processors increase so as to maintain a fixed efficiency. While this is not practical in general, because the memory per processor is limited, Kumar and Rao [13] proposed the isoefficiency scalability metric based on constant efficiency scaling. The *isoefficiency-scalability function* describes the rate at which the problem size should grow with the number of processors to maintain a fixed efficiency. Later in this chapter the details of the isoefficiency-scalability metric are examined.

Depending on the application area of the parallel program other notations of scaling are important. For example, scientific applications often simulating some physical phenomenon. Typically, scaling the problem size causes the amount of parallel work to be increased faster than a simple time-complexity analysis would predict so as to control the amount of simulation error [24].

2.3. Relevant Work

Some work has been done to try to relate several notions of scalability. The following are especially relevant since they involve relating memory-bounded speedup to other speedup notions. An extensive review of the scalability literature is reported in [18].

Worley [29] examined fixed-size, fixed-time, and memory-bounded speedup curves for simple algorithms used to approximate model linear partial differential equations (PDEs). He found that the fixed-time and the memory-bounded speedup curves could be drastically different depending on the algorithm and machine's interconnection topology.

Sun and Ni [26] studied fixed-size speedup, fixed-time speedup, and

memory-bounded speedup models. They derived two sets of formulations for these models. The first set are more detailed than the standard definitions since these incorporate uneven load balancing (and to a much lesser extent communication overhead). Their second set of formulations are simplified in that they assume negligible communication overhead, and the workload consists of a sequential part that is independent of system size and perfectly parallelizable parallel part (i.e., no load imbalancing). Using these simplified formulations, they show that the fixed-size and fixed-time speedup models are really special cases of the memory-bounded speedup model. However, in a practical sense this is meaningless because of the simplifying assumptions.

The isoefficiency-scalability metric of Kumar and Rao [19] has been widely accepted [10] [11]. Intuitively, the isoefficiency-scalability function describes the rate at which the problem size should grow with the number of processors to maintain a fixed efficiency. To derive the isoefficiency function let the parallel execution time for an individual processor be split into useful work, t_e, and time performing overhead, t_o. Then, $P*T_p(N) = P*(t_e+t_o) = T_1(N)+T_o$, where T_o is the sum of overhead for all processors. Therefore, the efficiency can be written as

$$\text{Efficiency}(P, N) = \frac{\text{Speedup(N, P)}}{P} = \frac{T_1}{T_p \times P} = \frac{T_1}{T_1 \times T_o} = \frac{1}{1+\frac{T_o}{T_1}} \tag{2.6}$$

To maintain a constant efficiency, T_1 must be proportional to T_o, or

$$T_1 = K\,T_o, \text{ where } K = E/(1-E) \tag{2.7}$$

In Chapter 4, several examples of applying the isoefficiency function are performed for the

three algorithms being considered in this study.

2.4. CMP Scalability Metric

The CMP (Constant-Memory-per-Processor) scalability metric proposed in this thesis has previous been described in [6] and [7]. Intuitively, the CMP-scalability metric describes the rate of growth of the memory-bounded speedup as a function of the number of processors. To determine the CMP-scalability metric for a particular parallel algorithm, let comp(N) be the number of computations for the algorithm, mem(N) be the total memory required by the algorithm, and let β = mem(N)/P be the local problem size in terms of memory used. Then the *CMP speedup* is defined as

$$\text{CMP Speedup}(P, \beta) = \frac{T_1(P,\beta)}{T_P(\beta)} \tag{2.8}$$

by substituting $\text{mem}^{-1}(\beta * P)$ for N in the traditional speedup formula (2.1). The *CMP-scalability function* is the function that describes the asymptotic growth of CMP speedup(β,P) as P goes to infinity. Because the $T_p(\beta)$ term depends on the specific parallel computer used, the CMP-scalability function captures both the algorithmic and architectural aspects in one metric.

Alternatively, the CMP-scalability metric can be thought of as the average number of sequential operations that can be performed per one parallel time step assuming memory-bounded scaling. It which case the CMP scalability is just

$$\text{CMP scalability} = \frac{\text{Sequential Time Complexity}}{\text{CMP Parallel Time Complexity}}, \tag{2.9}$$

where the "CMP Parallel Time complexity" must take into account the memory-bounded

scaling. For instance, a local computation on the order of mem(N)/P would be considered as a constant since it does not change as the number of processors and the problem size increases. In Chapter 4 the CMP-scalability metric is demonstrated on the three parallel algorithms in this study.

CHAPTER 3. PARALLEL ALGORITHMS

Parallel algorithms can vastly differ in their communication and computational requirements. Also, parallel machines can have very different computation and communication capabilities. As a consequence, the scalability of an algorithm needs to be determined on the basis of both the parallel computer and the algorithm. Three algorithms with varying degrees of communication and computational requirements are presented for a two-dimensional mesh topology and a hypercube topology. A load-and-store processor architecture is assumed for the processing elements. The algorithms are matrix multiplication (MM), Gauss-Jordan elimination (GJE) with partial pivoting, and Fast Fourier Transform (FFT). In addition to their varying degrees of communication and computational requirements, these algorithms were selected because they are commonly used in many applications.

For each of the algorithms and topologies considered, execution-time models are developed. The models are developed as parametric models so that the impact of speeding up computation speed and communication speed can be studied in Chapter 7. The models split the total parallel execution time ($T_p(N)$) into computation time, communication time, memory-access time, and other miscellaneous overhead. The models for the mesh topology are experimentally verified on a 16K processor MasPar MP-1 and 4K processor MasPar MP-2. Chapter 6 contains the details of the verification process. The MasPar MPL codes for these algorithms are included in Appendix B.

3.1. Communication Primitives

For each of the two topologies being consider, both the store-and-forward and cut-through routing schemes will be considered. The terminology described below is used when describing the communication time for each of the algorithms.

Let m be the number of words in the message, x be the distance between communicating processor (# of connections away), t_s be the startup time for the communication of the message, t_h be the per-hop time (i.e., the time delay for a word of the message to hop from one processor to its neighboring processor), and t_w be the per-word transmission time. Then the *store-and-forward* communication time for a message containing m words between two processors that are x hops apart is:

$$t_s + (t_w m + t_h)x \tag{3.1}$$

and the *cut-through/worm-hole* communication time for a message containing m words between two processors that are x hops apart is:

$$t_s + t_w m + t_h x \tag{3.2}$$

3.2. The Parallel Algorithms

3.2.1 Cannon's Matrix Multiplication

A parallel matrix multiplication algorithm attributed to Cannon [4] is considered. Two N x N matrices A and B are two-dimensionally block decomposed on a $\sqrt{P} \times \sqrt{P}$ processors array, or a mesh embedding on a hypercube. First, the algorithm involves shifting the submatrices of A and B (Figure 3.1 (a)), so that the diagonal submatrices of A are in the first column of the processor mesh and the diagonal submatrices of B are in the first row of

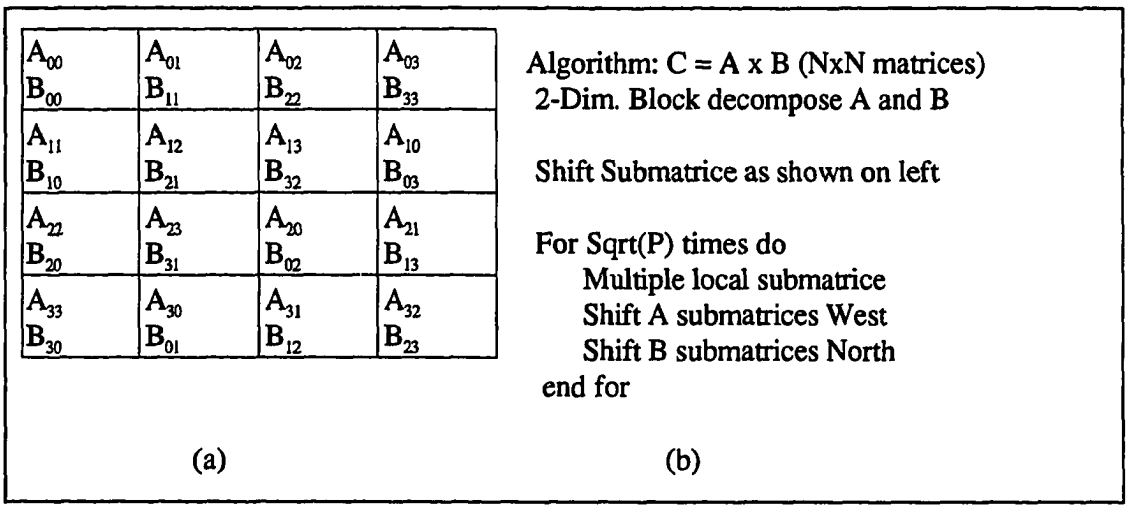

Figure 3.1. (a) Submatrices after initial shifting, (b) Cannon's Matrix multiplication algorithm on a two-dimensional mesh topology.

the processor mesh. The parallel algorithm (Figure 3.1 (b)) is performed in $\sqrt{P}$ steps with each step consisting of multiplying the local A and B submatrices at each processor, shifting the A submatrices west/left one processor, and shifting the B submatrices north/up one processor.

The parallel computation time (T_{COMP}^{MM}), communication time (T_{COMM}^{MM}), and memory-access time (T_{MEM}^{MM}) of Cannon's algorithm on a P processor two-dimensional mesh are

$$T_{COMP}^{MM} = \sqrt{P}\left[\left(\frac{N}{\sqrt{P}}\right)^3 (T_A + T_M)\right] \tag{3.3}$$

$$T_{COMM}^{MM} = 2\sqrt{P}\left[t_s + \left(\frac{N}{\sqrt{P}}\right)^2 t_w + t_h\right] \tag{3.4}$$

$$T_{MEM}^{MM} = \sqrt{P}\left[\left(\frac{N}{\sqrt{P}}\right)^2 (3T_{ST} + 2T_{LD}) + \left(\frac{N}{\sqrt{P}}\right)^3 (2T_{LD})\right] \tag{3.5}$$

where T_A is the time to perform an addition, T_M is the time to perform a multiplication, T_{ST} is

the time to perform a store operation, and T_{LD} is the time to perform a load operation. Since the communication is nearest neighbor on both topologies, the communication time is the same for store-and-forward and cut-through routing schemes.

3.2.2 Gauss-Jordan Elimination

To apply Gauss-Jordan Elimination (GJE) with partial pivoting to solve a linear system of equation, Ax = b, the N x N coefficient matrix A is two-dimensionally scatter decomposed on the $\sqrt{P} \times \sqrt{P}$ mesh of processors. On the hypercube topology, the mesh is embedded such that each row of the matrix is a subcube. The b vector is one-dimensionally scatter decomposed among the diagonal processors of the mesh. Partial pivoting is used for numerical accuracy.

Figure 3.2 outlines the parallel GJE algorithm. For each of the N pivot positions, the best pivot element is found, the pivot row is broadcast, the row multipliers are calculated, row multipliers are broadcast, and the rows are updated. The best pivot element is the

```
Data Layout: 2-D scatter decomposition of the N x N coefficient matrix, A

Parallel Algorithm:

For each N pivot elements do
   Find the best pivot element
   Broadcast pivot row
   Determine row multipliers
   Broadcast row multipliers
   Update rows
End for
```

Figure 3.2. Parallel Gauss-Jordan Elimination algorithm.

maximum element in the column of A containing the pivot element, which is called the pivot column. To find the best pivot element, processors storing pivot-column elements first search for their local best, and then these processors perform a parallel-prefix "sum" communication to find the global maximum. Broadcasting the pivot row and row multipliers are accomplished using cut-through routing across the full vertical dimension of the mesh. The row multipliers are calculated by the processors storing the pivot column. Updating the rows involve multiplying the row multiplier to the pivot row and adding this result to the existing row.

For the two-dimensional mesh topology, the parallel computation time (T_{COMP}^{GJE}), and memory-access time (T_{MEM}^{GJE}) formulas for GJE are

$$T_{COMP}^{GJE} = N[\frac{N}{\sqrt{P}}(2T_{COMPARE} + T_{NEG} + 2T_M + T_A) + \log_2\left(\sqrt{P}\right)T_{COMPARE} + \frac{N^2}{2P}(T_M + T_A) + T_D] + \frac{N}{\sqrt{P}}T_M \quad (3.6)$$

$$T_{MEM}^{GJE} = N[\frac{N}{2\sqrt{P}}(8T_{LD} + 7T_{ST}) + T_{LD} + T_{ST} + \frac{N^2}{2P}(2T_{LD} + T_{ST})] + \frac{N}{\sqrt{P}}(2T_{LD} + T_{ST}) \quad (3.7)$$

where $T_{COMPARE}$ is the time to perform a floating-point comparison, T_{NEG} *is* the time to perform a negation, and T_D is the time to perform a division. On the two-dimensional mesh topology the farthest processors containing a row or column of the matrix are $P^{1/2}$ hops away. Thus, the partial pivot-row and the column of row-multipliers must be broadcast a distance of $P^{1/2}$. Finding the best pivot element can be performed as a parallel-prefix "sum" computation, but the communication performed is a distance of $P^{1/2}$. The store-and-forward

communication time formula on the mesh topology is

$$T_{SF\_COMM}^{GJE,MESH} = N\left[t_s \log_2 \sqrt{P} + (t_w + t_h)\sqrt{p} + 2t_s + \tfrac{3}{2} t_w N + 2t_h \sqrt{P} \right] \tag{3.8}$$

and the cut-through communication formula is

$$T_{CT\_COMM}^{GJE,MESH} = N\left[t_s \log_2 \sqrt{P} + t_w + t_h \sqrt{p} + 2t_s + \tfrac{3}{2} t_w \tfrac{N}{\sqrt{P}} + 2t_h \sqrt{P} \right] \tag{3.9}$$

For the hypercube topology, only the communication time ($T_{COMM}^{GJE,HYPER}(P, N)$) formula differs from the two-dimensional mesh model. On the hypercube, the farthest processors that contain a row or column of the matrix are $\log_2 P^{1/2}$ hops away. This improves the broadcasting of the partial pivot-row and the column of row multipliers. Additionally, finding the best pivot element can perform a true parallel-prefix "sum" communication. The store-and-forward communication time formula on the hypercube is

$$T_{SF\_COMM}^{GJE,HYPER} = N\left[t_s \log_2 \sqrt{P} + (t_w + t_h)\log_2 \sqrt{p} + 2t_s + \tfrac{3}{2} t_w \tfrac{N}{\sqrt{P}} \log_2 \sqrt{P} + 2t_h \log_2 \sqrt{P} \right] \tag{3.10}$$

and the cut-through communication time formula on the hypercube is

$$T_{CT\_COMM}^{GJE,HYPER} = N\left[(t_s + t_w)\log_2 \sqrt{P} + t_h \log_2 \sqrt{P} + 2t_s + \tfrac{3}{2} t_w \tfrac{N}{\sqrt{P}} + 2t_h \log_2 \sqrt{P} \right] \tag{3.11}$$

3.2.3. Fast Fourier Transform, FFT

The Discrete Fourier Transform (DFT) of an N-point sequence $<a_k>$, $0 \le k < N$, is another N-point sequence $<A_m>$, $0 \le m < N$, defined as

$$A_m = \sum_{k=0}^{N-1} a_k \omega_N^{km}, \quad 0 \le m < N, \tag{3.12}$$

where ω_N is a primitive N^{th} root of unity, i.e., $\omega_N = e^{-i(2\pi/N)}$. Fast Fourier Transform (FFT)

algorithms are efficient (serial O(N log N)) methods for calculating the DFT. The specific FFT algorithm considered here is the radix-two decimation-in-frequency (DIF) algorithm [4] where N is a power of two.

The radix-two DIF algorithm is a divide-and-conquer algorithm which divides the N-point sequence $<a_k>$ into two sequences $<b_h>$ and $<c_h>$. The sequence $<b_h>$, $0 \leq h < N/2$, is equal to the first half of $<a_k>$, i.e., $<b_h> = <a_h>$, $0 \leq h < N/2$, and $<c_h>$, $0 \leq h < N/2$, is equal to the second half of $<a_k>$, i.e., $<c_h> = <a_{h+N/2}>$, $0 \leq h < N/2$. The DFT of the N-point sequence $<a_k>$ is then computed in terms of the two N/2-point DFTs of the sequences $<b_h + c_h>$, $0 \leq h < N/2$, and $<(b_h - c_h)\omega_N^h>$, $0 \leq h < N/2$. This process is repeated $\log_2 N$ times.

Figure 3.3a illustrates the radix-two DIF algorithm for a 32-point DFT computation. The basic computation is the butterfly operation (denoted by the open dots), as shown in Figure 3.3b. Each stage has 16 butterfly operations. The numbers along the left-hand side of Figure 3.3a represent the initial $<a_k>$ input sequence. For example, butterfly operation $B_{0,0}$ in stage 0 takes the 0^{th} and 16^{th} components of the $<a_k>$ sequence as input. The numbers along the right-hand side represent the output sequence, $<A_m>$. Each butterfly operation uses a twiddle factor which is a power of the primitive N^{th} root of unity (ω). The necessary twiddle factor is shown below each butterfly operation in Figure 3.3a.

A *binary-exchange* FFT algorithm based on the above radix-two DIF algorithm is described next for the two-dimensional mesh and hypercube topologies. The algorithm is called binary-exchange because at different stages of the FFT algorithm two processors

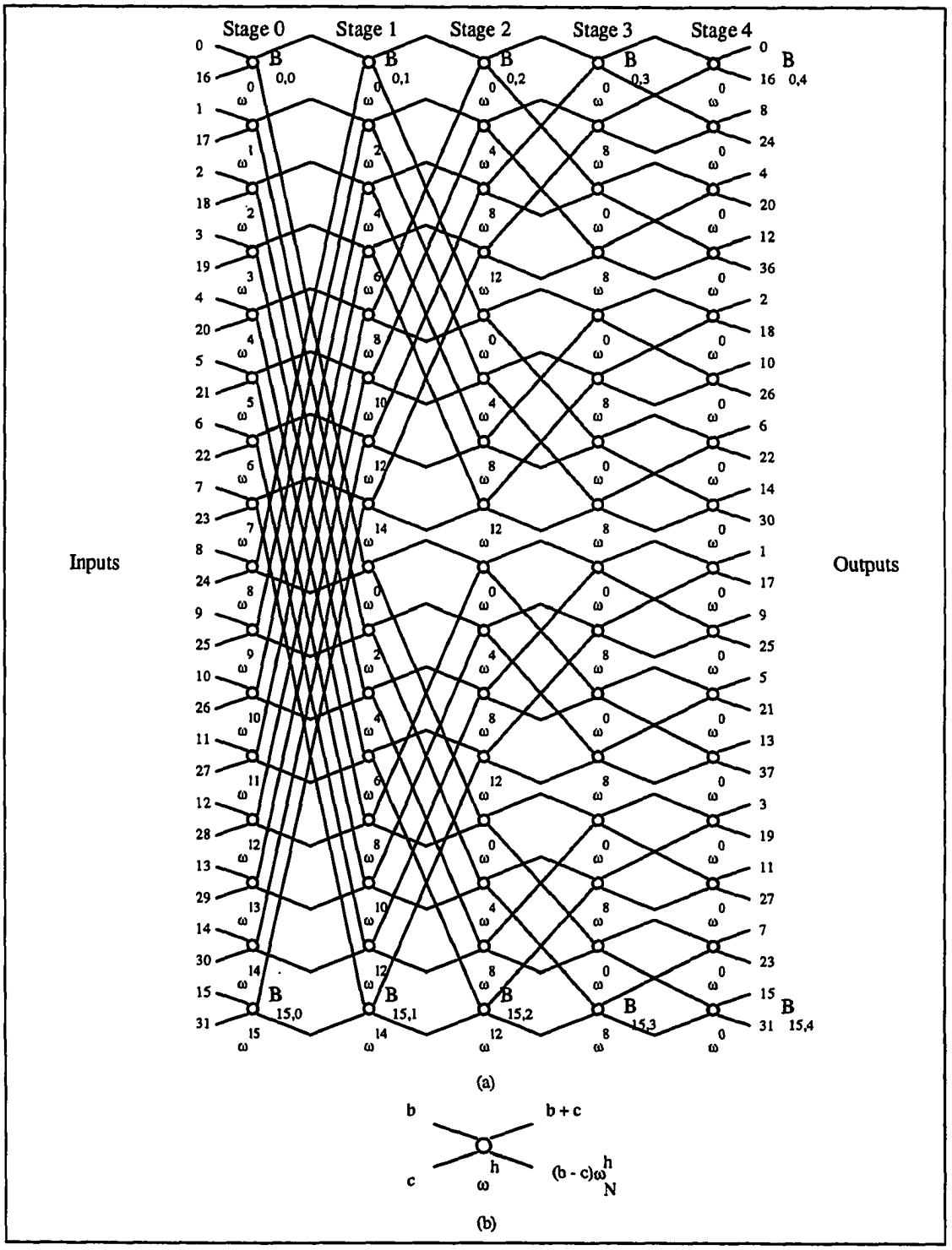

Figure 3.3. (a) Computation of 32-point DFT. (b) Radix-two butterfly operation.

```
Data layout: One-dimensionally scattered
Parallel Algorithm:
For log2N stages do
    Compute local butterflies
    Update Twiddles
    If necessary, exchange elements
end for
```

Figure 3.4. Outline of the parallel FFT algorithm.

exchange information only if each differs by a single binary bit in their processor number. The general outline of the parallel FFT algorithm is given in Figure 3.4.

When implementing the FFT algorithm, the following questions must be addressed: (1) What layout of the data and corresponding butterfly operations should be used?, (2) How can the necessary twiddle factors be supplied to a processor when needed?, (3) What is the best order of evaluation for butterfly operations, (4) How should elements to be communicated be blocking?, and (5) How can the memory access penalty load-and-store processors be avoided? These issues are not orthogonal and involve various tradeoffs that are machine specific. For example, the MasPar FFT implementation discussed in Chapter 5 describes how to modify the order of the butterfly operations so as to reduce the number of memory accesses. This particular optimization makes use of the fine-grained communication network on the MasPar architecture, and would not be effective on medium or course-grained parallel computers. In the remainder of this chapter the FFT implementation on a "generic" mesh machine is described to avoid machine specific details.

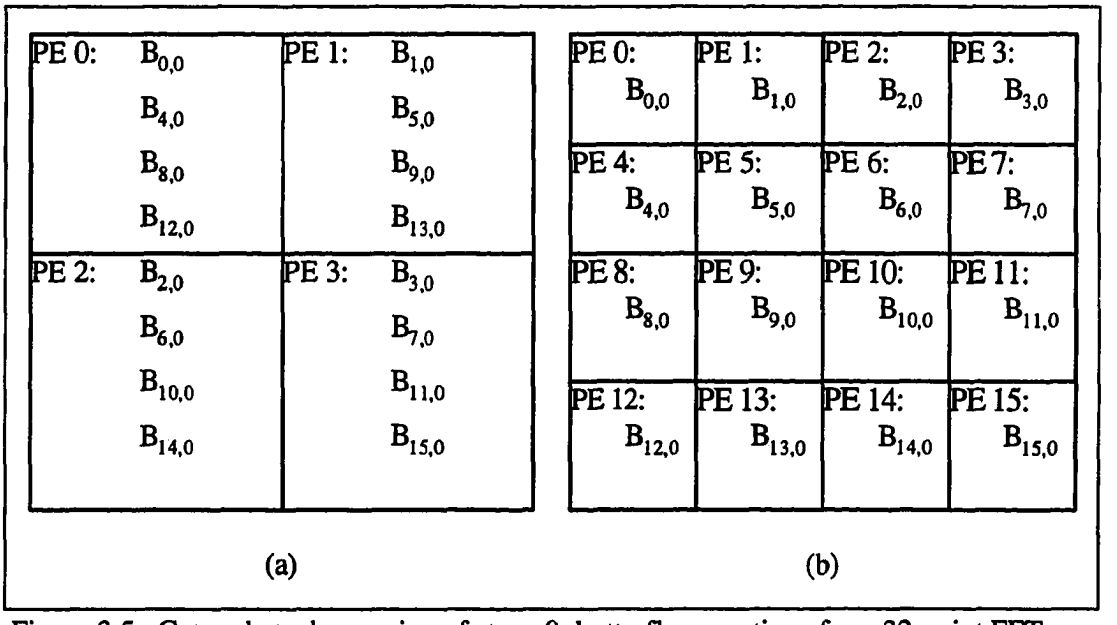

Figure 3.5. Cut-and-stack mapping of stage 0 butterfly operations for a 32-point FFT on (a) a 2x2 PE array and (b) a 4x4 PE array.

On a two-dimensional mesh topology, the butterfly operations need to be mapped to the PE array to minimize the performance loss due to idle processors and the communication overhead. A "cut-and-stack" approach that assigns every P^{th} butterfly operation from each stage to the same processor, where P is the number of processors, works best for a mesh topology. For example, a 32-point FFT on a 2x2 PE array will have four butterfly operations per PE as shown in Figure 3.5a. This layout minimizes the communication overhead as follows. To begin with, the first $\log_2(N/P)$ stages, called the *in-memory stages*, require no inter-PE communication since pairs of communicating butterflies reside on the same PE. The remaining $\log_2(P)$ stages require inter-PE communication. However, with this data layout communication is only necessary between

processors within the same row or column. To see this, consider Figure 3.5b which shows the cut-and-stack butterfly layout for a 32-point FFT where only the first stage is an in-memory stage.

After the in-memory stage, each PE in the upper half of the PE array communicates with the PE two processors below it in the same column. For example, the bottom output of the butterfly operation $B_{0,0}$ at PE 0 is exchanged with the top output of the butterfly operation $B_{8,0}$ at PE 8. In general, letting dim_x and dim_y represent the x and y dimensions of the PE array, the communication pattern for successive communication stages involves PEs exchanging data elements a distance of $dim_y/2$, $dim_y/4$, ..., 2, 1 along the y-dimension followed by exchanging data elements a distance of $dim_x/2$, $dim_x/4$, ..., 2, 1 along the x-dimension.

A *stage-at-a-time evaluation* is assumed for the general algorithm, where each PE performs all of the butterfly operations for a stage before starting on the butterfly operations for the next stage. Between the communication stages, a single block of data can be exchanged between communicating processors.

The following scheme for supplying the twiddle factors for the butterfly operations is assumed. Before stage 0 of the computation, the initial twiddle factors are calculated using the sine and cosine functions via the equation

$$\omega_N^k = \cos(2\pi k/N) + i \sin(2\pi k/N). \qquad (3.13)$$

Recalculation of twiddle factors utilizing equation (3.13) before each stage would be time consuming. Here, one alternative to recalculating twiddle factors from (3.13), called the

square-of-twiddles, is assumed. The square-of-twiddles option makes use of the modular nature of the twiddle factors to generate a stage's (except the initial stage) twiddle factors from the previous stage's twiddle factors [5]. The square-of-twiddles method squares a butterfly's twiddle factor and selectively negates it if the resulting power is greater than N. This method is substantially faster than calculating the twiddle factors via equation (3.13) since multiplication and negation are much faster than calling the sine and cosine functions. One drawback of the square-of-twiddles method is the accumulation of round-off errors incurred by the repeated squaring of the twiddle factors, but using double precision numbers for the twiddle factors can overcome this problem.

For an N-point FFT on a $\sqrt{P}$ x $\sqrt{P}$ processor mesh, the parallel computation time (T_{COMP}^{FFT}), and memory-access (T_{MEM}^{FFT}) are given by

$$T_{COMP}^{FFT} = \frac{N}{2P}[T_W + 8(T_M + T_A)\log_2(N)] \tag{3.14}$$

$$T_{MEM}^{FFT} = \frac{N}{2P}[6T_{LD} + 6T_{ST} + 6(T_{LD} + T_{ST})\log_2(\frac{N}{P})] \tag{3.15}$$

where T_W is the time to calculate an initial twiddle factor by (3.13). The communication scheme is limited to store-and-forward routing since contention between communicating processors prevents improvement when using cut-through routing. To see this consider the example shown in Figure 3.5b. In the first communication stage, processors in the first row must exchange blocks of data with processors in the third row, while processors in the second row must exchange blocks of data with processors in the fourth row. Thus, the communication time is given by

$$T_{COMM}^{FFT,MESH} = 2t_s \log_2 P + 4\sqrt{P}\left(t_w \frac{N}{P} + t_h\right) \tag{3.16}$$

For the hypercube topology, only the communication time formula differs from the two-dimensional mesh model. Upon embedding the mesh into a hypercube, the communication time is decreased, since each pair of communicating processors at all communication stages are directly connected. For directly connected processors, store-and forward and cut-through routing give the same performance, so the hypercube communication time is given by

$$T_{COMM}^{FFT,HYPER} = 2\log_2 P\left(t_s + t_w \frac{N}{P} + t_h\right). \tag{3.17}$$

CHAPTER 4. ASYMPTOTIC ANALYSIS OF SCALABILITY

In this chapter the MM, GJE, and FFT execution-time formulas, described in Chapter 3, for the two-dimensional mesh and hypercube topologies are used to demonstrate how the asymptotic CMP-scalability and isoefficiency-scalability metrics can be calculated. Apparently conflicting scalability results between the CMP and isoefficiency-scalability metrics for Gauss-Jordan Elimination and Fast Fourier Transform on a mesh are identified.

This apparent conflict is resolved by observing that these parallel algorithms each describe a performance surface over the plane of two independent variables: the number of processors and the problem size. The different scaling assumptions of the CMP and isoefficiency-scalability metrics describe different planar cross-sections in this three-dimensional space. The CMP-scalability and isoefficiency functions correspond to the intersections of these planes with the performance surface. Thus, the CMP and isoefficiency functions actually provide complimentary information about the performance surfaces and not conflicting information.

Using the complimentary information from both the CMP and isoefficiency functions, two theorems are proven that predict the relative change in performance if only the number of processors is varied, or if only the size of the problem is varied.

This chapter concludes with an examination of why most algorithms are not fixed-time scalable. Two classes of algorithms are identified that are shown not to be fixed-time scalable. These two classes of algorithms include the vast majority of algorithms.

4.1. Framework for Asymptotic Analysis

4.1.1. Terminology

Let N be the problem size, P be the number of processors, $T_1(N)$ be the sequential execution time using one processor, and $T_P(N)$ be the parallel execution time using P processors. Then

$$\text{Speedup}(P, N) = \frac{T_1(N)}{T_P(N)} \tag{4.1}$$

$$\text{Efficiency}(P,N) = \frac{\text{Speedup}(P,N)}{P} = \frac{T_1(N)}{P*T_P(N)} \tag{4.2}$$

The execution time of the parallel algorithm can be split into the time spent performing computation $(T_{COMP}(P, N))$, communication $(T_{COMM}(P, N))$, other miscellaneous overhead $(T_{MISC}(P, N))$, and memory accesses $(T_{MEM}(P, N))$, i.e.,

$$T_P(P, N) = T_{COMP}(P, N) + T_{COMM}(P, N) + T_{MISC}(P, N) + T_{MEM}(P, N) \tag{4.3}$$

Ideally , $(T_{COMP}(P, N) + T_{MISC}(P, N) + T_{MEM}(P, N)) = T_1(N)/P$. However, parallelization often introduces additional inefficiencies, such as imperfect load-balancing, that causes $(T_{COMP}(P, N) + T_{MISC}(P, N) + T_{MEM}(P, N)) > T_1(N)/P$.

A *reasonable parallelization* is defined to be a parallel implementation of an algorithm such that $(T_{COMP}(P, N) + T_{MISC}(P, N) + T_{MEM}(P, N)) = C_O*T_1(N)/P$, where C_O is a constant. In other words, a constant slowdown in the parallel implementation due to non-communication overhead is reasonable. (Some factors could cause C_O to be less than one, e.g., memory accesses closer to the processors than in the sequential algorithm.) The speedup and efficiency formulas assuming reasonable parallelization become:

$$\text{Speedup}(P, N) = \frac{PT_1(N)}{C_O T_1(N) + PT_{comm}(P,N)}, \tag{4.4}$$

and

$$\text{Efficiency}(P, N) = \frac{T_1(N)}{C_O T_1(N) + PT_{comm}(P,N)}, \tag{4.5}$$

where C_O is a constant representing the slowdown due to non-communication overhead. Since we are concerned with only asymptotic scalability metrics, machine specific constants, such as processor speed and communication speed, can be ignored.

4.1.2. Isoefficiency Scalability Metrics

The scaling assumption in isoefficiency is that a fixed machine efficiency is maintained as the number of processsors, P, increases. To accomplish this, the problem size, N, generally increases faster that the rate of processors. The function expressing the rate at which the problem size must grow with respect to the number of processors is called the isoefficiency function. Thus, for a specific isoefficiency function, say $f(P)$, the Efficiency(P, N) is a constant K as $P \rightarrow \infty$. Dividing numerator and denominator of (4.5) by $T_1(N)$ gives

$$\text{Efficiency}(P, N) = \frac{1}{C_O + PT_{comm}(P,N)/T_1(N)} = K \tag{4.6}$$

It is easy to see that $O(PT_{comm}(P, N)) <= O(T_1(N))$ as $P \rightarrow \infty$.

The procedure for determining the isoefficiency function is:

1) examine each term, say t, in $PT_{comm}(P, N)$ finding $N = f(P)$, such that substituting for P in t results in $O(T_1(N))$,

2) from all the terms in $PT_{comm}(P, N)$ select the $N=f(P)$ function that grows fastest in terms of P, and

3) determine the isoefficiency in terms of sequential work by substituting the $f(P)$ function selected in step 2) into $T_1(N)$.

The above process describes the isoefficiency function as intended by Kumar and Rao [19], but a similar procedure can be done to determine the isoefficiency in terms of memory usage. Call this modified isoefficiency metric, *memory-isoefficiency*. The steps for determining the memory-isoefficiency function are:

1) rewrite $T_1(N)$ and $T_{comm}(P, N)$, so the N represents the total problem size. Let $T_1'(N)$ and $T'_{comm}(P, N)$ denote these revised formulas,

2) examine each term, say t, in $PT'_{comm}(P, N)$ finding $N = f(P)$, such that substituting for P in t results in $O(T_1(N))$, and

3) from all the terms in $PT'_{comm}(P, N)$ select the $N=f(P)$ function that grows fastest in terms of P. This is the memory-isoefficiency function.

4.1.3. Fixed-time Scalability Metrics

In fixed-time scaling the problem size is allowed to grow as the number of processors increases so as to maintain some constant overall execution time. In otherwords, determine $N = f(P)$ such that $T_p(f(P))$ is constant as $P \rightarrow \infty$. Thus, each term in $T_p(N)$ converges to a constant as $P \rightarrow \infty$. As will be demonstrated, most parallel systems are not asymptotically scalable under fixed-time scaling.

The procedure for determine the fixed-time scalability function is:

1) examine each term, say t, in $T_p(N)$ finding $N = f(P)$, such that t converges to a constant as $P \rightarrow \infty$. If some term does not converge to a constant, then the parallel system is not

fixed-time scalable, and

2) determine the *fixed-time-scalability function* by selecting the $N=f(P)$ function that grows slowest in terms of P since this forces all the other terms to zero as $P \rightarrow \infty$.

Later in this chapter, two classes of algorithms are defined that are not fixed-time scalable as described in step (1).

4.1.4. CMP Scalability Metrics

The scaling assumption in CMP scalability is that the global problem size grows linearly with the number of processors, i.e., the local problem size per processor is fixed, so $N = f(P)$ is fixed. As described in Chapter 2, the CMP-scalability function is $O(\text{Speedup}(P, N))$ under memory-bounded scaling as $P \rightarrow \infty$.

The procedure for determining the CMP scalability function is:

1) start with the traditional speedup formula (4.4) and apply memory-bounded scaling:

$$\text{Speedup}(P, N) = \text{Speedup}(P, f(P)) = \frac{PT_1(f(P))}{C_O T_1(f(P)) + PT_{comm}(P,N)} \tag{4.7}$$

2) determine the CMP-scalability function by finding the big-oh complexity of (4.7).

4.2. Asymptotic Scalability Analysis of MM, GJE, FFT

4.2.1. MM, Matrix Multiplication

The matrix multiplication algorithm uses only nearest neighbor communication for both the two-dimensional mesh and hypercube topologies so the type of topology does not effect the scalability. Additionally, cut-through routing and store-and-forward routing provide the same performance since nearest neighbor communication is utilized.

Following the procedure outlined in subsection 4.1.2, the isoefficiency of matrix multiplication is determined as followed:

Step 1) The sequential execution time is $O(T_1(N)) = N^3$. From equation 2.4, the asymptotically important terms of $T_{COMM}(P, N)$ are $\sqrt{P} + N^2/\sqrt{P}$, so each term in $PT_{COMM}(P, N) = P^{3/2} + \sqrt{P}N^2$ must be examined. For the $\sqrt{P}N^2$ term, $N^3 \propto \sqrt{P}N^2$ so $N \propto P^{1/2}$. For the $P^{3/2}$ term, $N^3 \propto P^{3/2}$, so $N \propto P^{1/2}$.

Step 2) Both terms in step (1) give $N \propto P^{1/2}$, so clearly $P^{1/2}$ grows the fastest in terms of P.

Step 3) Substituting $P^{1/2}$ for N in $T_1(N)=O(N^3)$ results in an the isoefficiency function of $O(P^{3/2})$.

Following the procedure outlined in subsection 4.1.3, the fixed-time scalability of matrix multiplication is determined as followed:

Step 1) Upon examining the $\sqrt{P}$ communication term of the $T_p(N)$, equation 2.4, it clearly does not converge as $P \rightarrow \infty$, so matrix multiplication is not fixed-time scalable.

Following the procedure outlined in subsection 4.1.3, the fixed-time scalability of matrix multiplication is determined as followed:

Step 1) For matrix multiplication, $N = f(P) = \sqrt{P}$ under memory-bounded scaling since N^2 elements are scattered over P processors. The speedup formula under memory-bounded scaling for matrix multiplication is

$$\text{Speedup}(P, f(P)) = \frac{PT_1(f(P))}{C_O T_1(f(P)) + PT_{comm}} = \frac{PP^{3/2}}{P^{3/2} + P(P^{1/2} + t_w)} \qquad (4.8)$$

Step 2) The complexity of the (4.8) determines the CMP scalability function to be $O(P)$.

4.2.2 GJE, Gauss-Jordan Elimination

The run-time of GJE differs depending on the processor topology and the routing algorithm. The scalability of GJE on a mesh topology assuming store-and-forward routing is demonstrated. Remaining combinations of topologies and routing algorithms are summarized in Table 4.1, Table 4.2, and Table 4.3. Following the procedure outlined in subsection 4.1.2, the isoefficiency of Gauss-Jordan elimination is determined as follows:

Step 1) The sequential execution time *is* $O(T_1(N)) = N^3$. From equation 2.8, asymptotically important terms of $T_{COMM}(P, N)$ are $N\log_2\sqrt{P} + N\sqrt{P} + N + N^2$. Therefore, the terms in $PT_{COMM}(P, N)$ on a mesh using store-and-forward are $PN\log_2\sqrt{P} + NP^{3/2} + NP + N^2P$. For the $PN\log_2\sqrt{P}$ term, $N^3 \propto PN\log_2\sqrt{p}$ so $N \propto \sqrt{P\log_2\sqrt{P}}$. For the $NP^{3/2}$ term, $N^3 \propto NP^{3/2}$ so $N \propto P^{3/4}$. For the NP term, $N^3 \propto NP$ so $N \propto P^{1/2}$. For the N^2P term, $N^3 \propto N^2P$ so $N \propto P$.

Step 2) The $N \propto P$ term grows the fastest in terms of P, so it is selected.

Step 3) Therefore, the isoefficiency function of GJE on a mesh with store-and-forward routing is $O(P^3)$.

To determine the fixed-time scalability of GJE, the procedure outlined in subsection 4.1.3 is followed. Upon examining the terms in $T_p(N)$ during step (1), several of the communication terms in equation 2.8 will grow without bounds as $P \rightarrow \infty$. Therefore, it is concluded that Gauss-Jordan elimination is not fixed-time scalable.

The CMP scalability of GJE is determined by following the procedure outlined in

Table 4.1: Summary of isoefficiency results.

Communication Topology	Routing Scheme	Matrix Multiplication	Gauss-Jordan Elimination	Fast Fourier Transform
Mesh	store-and-forward	$O(P^{3/2})$	$O(P^3)$	$O(P^{1/2}2^P)$
	cut-through		$O(P^{9/4})$	
Hypercube	store-and-forward		$O(P^{3/2}(\log_2 P)^3)$	$O(P\log_2 P)$
	cut-through		$O(P^{3/2}(\log_2 P)^{3/2})$	

Table 4.2: Summary of fixed-time scalability results.

Communication Topology	Routing Scheme	Matrix Multiplication	Gauss-Jordan Elimination	Fast Fourier Transform
Mesh	store-and-forward	Not scalable due to the sqrt(P) communication term	$O(P^{-1/2})$ not scalable	Not scalable due to the sqrt(P) communication term
	cut-through		$O(P^{-1/2})$ not scalable	
Hypercube	store-and-forward		$O(1/(\log_2 P))$ not scalable	Not scalable due to the $\log_2 P$ communication term
	cut-through		$O(1/(\log_2 P))$ not scalable	

Table 4.3: Summary of CMP scalability results.

Communication Topology	Routing Scheme	Matrix Multiplication	Gauss-Jordan Elimination	Fast Fourier Transform
Mesh	store-and-forward	O(P)	$O(P^{1/2})$	$O(P^{1/2}\log_2 P)$
	cut-through		$O(P^{1/2})$	
Hypercube	store-and-forward		$O(P/\log_2 P)$	O(P)
	cut-through		$O(P/\log_2 P)$	

subsection 4.1.4. The steps are:

Step 1) For GJE, $N = f(P) = \sqrt{P}$ under memory-bounded scaling since N^2 elements are distributed over P processors. The speedup formula for GJE under memory-bounded scaling is

$$\text{Speedup}(P, f(P)) = \frac{PT_1(f(P))}{C_O T_1(f(P)) + PT_{comm}(P, f(P))} = \frac{PP^{3/2}}{P^{3/2} + P(\sqrt{P}\log_2 P + \sqrt{P} + P)} \quad (4.9)$$

Step 2) The complexity of equation 4.9 shows the CMP-scalability function of GJE to be $O(\sqrt{P})$.

4.2.3. FFT, Fast Fourier Transform

The scalability of FFT differs depending on the processor topology. The routing scheme does not effect the scalability. On the mesh topology, cut-through routing does not improve the scalability because of contention for the communication links. On the hypercube topology, cut-through and store-and-forward routing offer the same scalability since all communications are between directly connected processors. The scalability of FFT on a mesh topology is demonstrated. The scalability of the hypercube topology is summarized in Table 4.1, Table 4.2, and Table 4.3.

The isoefficiency scalability of FFT is determined by following the procedure outlined in subsection 4.1.2, which is:

Step 1) The sequential execution time *is* $O(T_1(N)) = N \log_2 N$. From equation 2.15, asymptotically important terms of $T_{COMM}(P, N)$ are $\log_2 P + N/\sqrt{P}$. Therefore, the terms in $PT_{COMM}(P, N)$ on a mesh topology are $P\log_2 P + N\sqrt{P}$. For the $P \log_2 P$ term, $N \log_2 N \propto$

$P\log_2 P$, so $N \propto P$. For the $N/\sqrt{P}$ term, $N\log_2 N \propto N\sqrt{P}$, so $N \propto 2^{\sqrt{P}}$.

Step 2) The $2^{\sqrt{P}}$ term grows the fastest in terms of P, so it is selected.

Step 3) Therefore, the isoefficiency function of FFT on a mesh is $O(\sqrt{P}\,2^{\sqrt{P}})$.

By following the procedure outlined in subsection 4.1.3, the fixed-time scalability of Fast Fourier Transform is determined as follows:

Step 1) Upon examining the terms in $T_p(N)$, several of the communication terms in equation 2.15 with grow without bounds as $P \rightarrow \infty$, so Fast Fourier Transform is not fixed-time scalable.

Finally, the CMP scalability of FFT is determined. By following the procedure outlined in subsection 4.1.4., the steps in calculating the CMP scalability are:

Step 1) For FFT, $N = f(P) = N$ under memory-bounded scaling since N elements are distributed over P processors. The speedup formula for FFT under memory-bounded scaling is

$$\text{Speedup}(P, f(P)) = \frac{PT_1(f(P))}{C_O T_1(f(P)) + PT_{comm}(P, f(P))} = \frac{PP\log_2 P}{P\log_2 P + P(\log_2 P + \sqrt{P})} \quad (4.10)$$

Step 2) The complexity of equation 4.10 shows the CMP scalability function of FFT to be $O(\sqrt{P}\log_2 P)$.

4.3. Conflicting Predictions of Isoefficiency and CMP Scalability for GJE and FFT

The asymptotic isoefficiency and CMP-scalability results for GJE and FFT on a mesh topology appear to contradict. As illustrated in Figures 4.1 and 4.2, the memory-isoefficiency metric predicts that the GJE algorithm on a mesh will scale better than

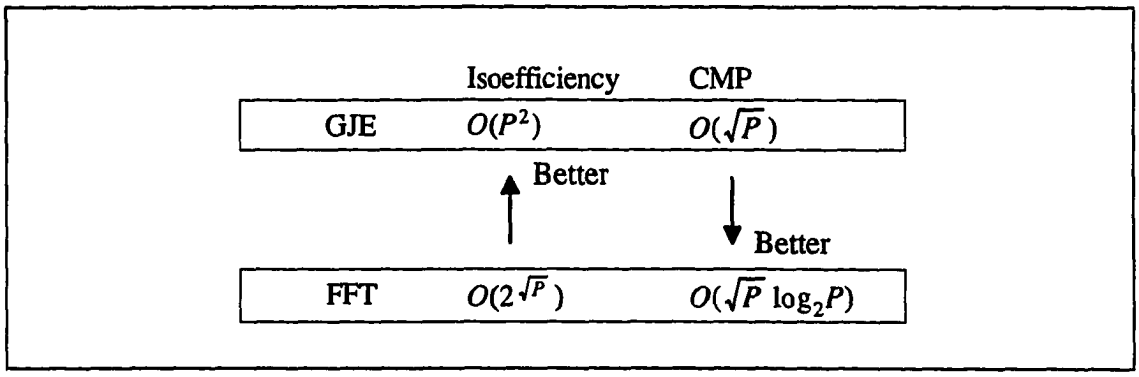

Figure 4.1. Comparison of memory-isoefficiency and CMP scalability for GJE and FFT on a mesh using store-and-forward routing.

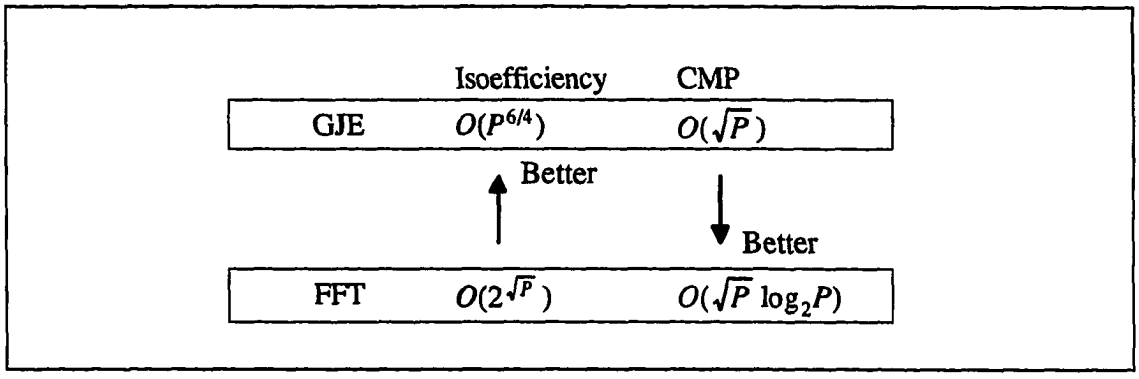

Figure 4.2. Comparison of memory-isoefficiency and CMP scalability for GJE and FFT on a mesh using cut-through routing.

the FFT algorithm, whereas the CMP metric predicts that the FFT algorithm will scale better than the GJE algorithm. Recall that a slower growing isoefficiency function indicates better scalability than a faster growing function, while the reverse is true for CMP scalability functions.

To explain this apparent contradiction, it must be remembered that the isoefficiency and CMP-scalability metrics have different scaling assumptions. In isoefficiency, the local

problem size per processor is allowed to increase so as to maintain a constant machine efficiency as the number of processors increase. However, in CMP scalability the local problem-size per processor is held constant and the machine efficiency is allowed to degrade as the number of processors increase. The complexity of the efficiency under CMP scaling is

$$\text{CMP efficiency} = O\left(\frac{\textit{CMP scalability function}}{P}\right) \tag{4.11}$$

Table 4.4 contains the complexity of CMP efficiency for MM, GJE, and FFT on mesh and hypercube topologies. This shows that asymptotically the efficiency for the FFT will eventually be better than the GJE for the topologies and routing methods considered.

Another way of viewing the isoefficiency and CMP-efficiency scalability metrics is as different planar cross-sections of the Efficiency(P, N) surface for a particular parallel algorithm-machine pair. For example, the Efficiency(P, N) surface for Gauss-Jordan Elimination is shown in Figure 4.3. The isoefficiency scaling assumption maintains a constant efficiency, say efficiency = e, that describes a horizontal plane in the three-dimensional space. As shown in Figure 4.4, the isoefficiency function for efficiency = e describes the intersection between the efficiency surface with the horizontal plane where efficiency = e. Actually, there

Table 4.4. Complexity of CMP efficiency.

Communication Topology	Routing Scheme	Matrix Multiplication	Gauss-Jordan Elimination	Fast Fourier Transform
Mesh	store-and-forward	O(1)	$O(P^{-1/2})$	$O(P^{-1/2}\log_2 P)$
	cut-through		$O(P^{-1/2})$	
Hypercube	store-and-forward		$O(1/\log_2 P)$	O(1)
	cut-through		$O(1/\log_2 P)$	

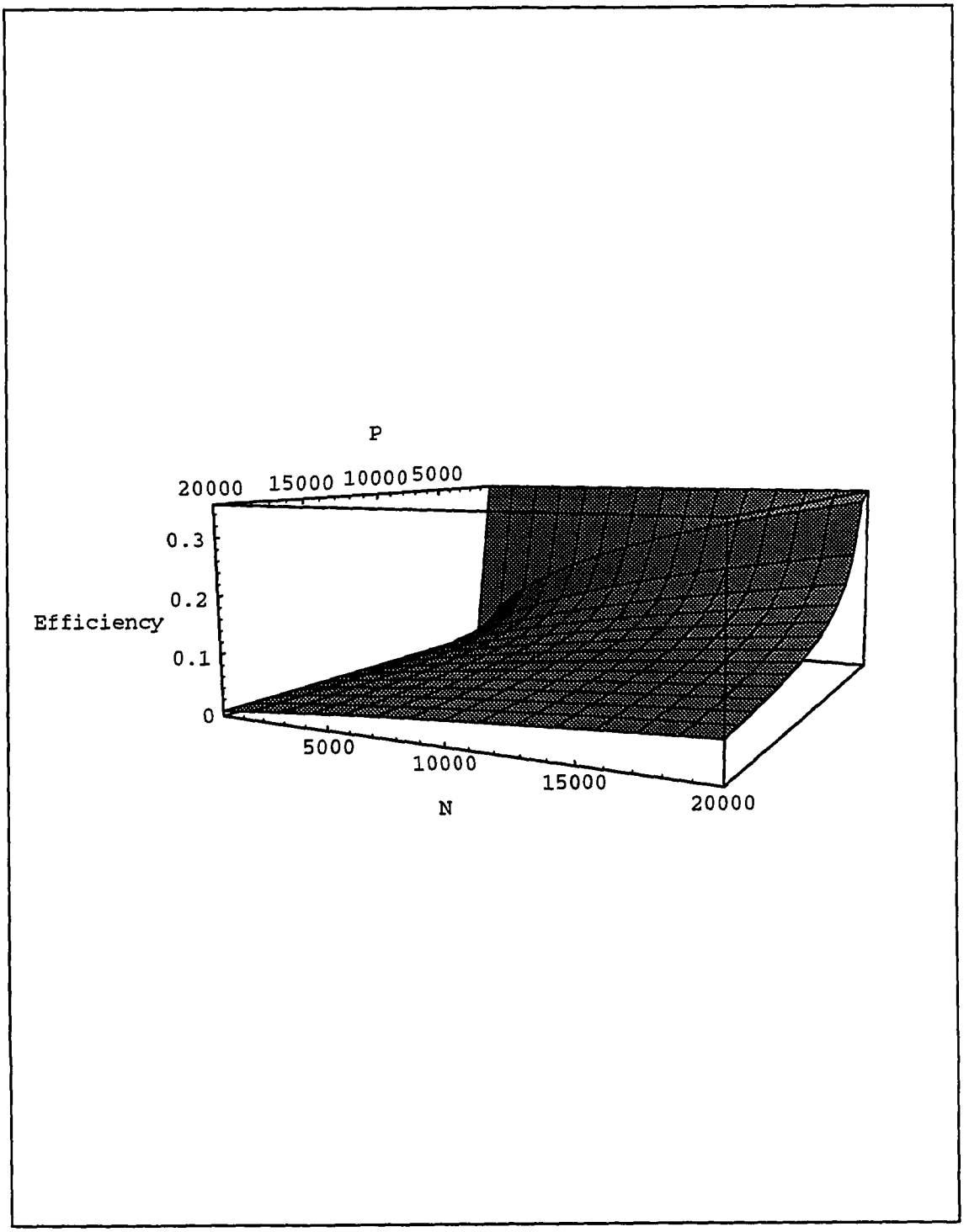

Figure 4.3. Gauss-Jordan Elimination efficiency surface on the MasPar MP-1.

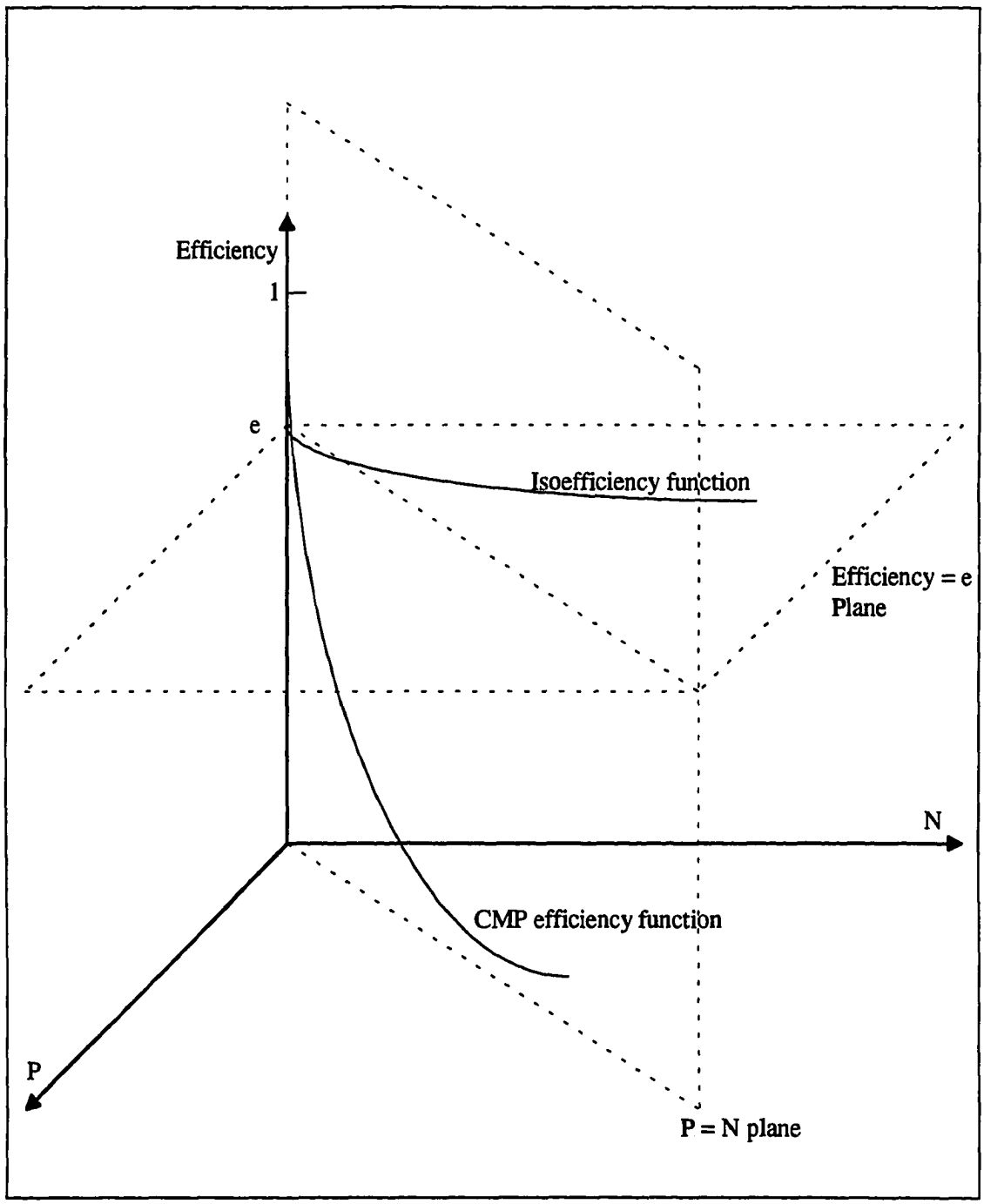

Figure 4.4. The isoefficiency function and CMP-efficiency function as planar cross sections of the efficiency surface.

is a whole family of isoefficiency functions corresponding to intersection of the efficiency surface for the algorithm with horizontal planes for each efficiency value.

The CMP-scaling assumption fixes N/P = at some constant β. A specific constant β describes a plane perpendicular to the efficiency planes passing through the origin. The CMP-efficiency function describes the intersection between the efficiency surface and the plane where $N/P = \beta$. Figure 4.4 shows the CMP-efficiency function for the intersection between the efficiency surface and the $\beta = 1$ plane where $P = N$.

4.4. Using the Complimentary Information Provided by the Isoefficiency and CMP-Scability Functions

When two different algorithms being compared have conflicting scaling information from the isoefficiency and CMP-scalability functions, their relative change in performance under fixed-machine (fixed-processor) size scaling and under fixed-problem size scaling can be inferred. In fixed-machine scaling, the number of processors is fixed and the problem size is increased, while fixed-problem size scaling fixes the problem size and increases the number of processors. Theorem 4.1 describes how the efficiency of the two algorithms will change under fixed-problem size scaling, and Theorem 4.2 describes how the efficiency of the two algorithms will change under fixed-machine size scaling.

Theorem 4.1. Let $E(A, P, N)$ be the efficiency when algorithm A is executed on P processors for a problem size of N. Let p_1 and p_2 be the number of processors that lead to the same efficiency e_3 on a problem size of n_1 for algorithms A_1 and A_2 respectively, i.e., $E(A_1, p_1, n_1) = E(A_2, p_2, n_1) = e_3$. Let the isoefficiency functions $Iso_1(P)$ and $Iso_2(P)$ for algorithms A_1 and A_2 be such that $\Theta\ (Iso_1(P)) > \Theta\ (Iso_2(P))$. Let the CMP-scalability

functions $CMP_1(P)$ and $CMP_2(P)$ for algorithms A_1 and A_2 be such that $\Theta(CMP_1(P)) > \Theta(CMP_2(P))$. Then, an increase in the number of processors for both algorithms to p_3, where $p_3 > p_2$ and $p_3 > p_1$, while keeping the problem size fixed at n_1, will cause a greater drop in efficiency for algorithm A_2 than for A_1. That is, for $p_3 > p_2$ and $p_3 > p_1$, $E(A_1, p_3, n_1) > E(A_2, p_3, n_1)$.

Proof: Consider the fixed-efficiency plane such that efficiency = e_3, where e_3 is a constant. The $Iso_1(P)$ and $Iso_2(P)$ functions describe how the efficiency surfaces of algorithm A_1 and algorithm A_2 are cut by this plane. Give that $\Theta(Iso_1(P)) > \Theta(Iso_2(P))$, this implies $p_1 < p_2$. Let α_1 be the point (e_3, p_1, n_1) and α_2 be the point (e_3, p_2, n_1). Figure 4.5 shows this situation.

Now, consider the CMP cross-section plane $\beta P = N$, where $\beta = n_1 / p_3$. The $CMP_1(P)$ and $CMP_2(P)$ functions describe how the efficiency surfaces of algorithm A_1 and algorithm A_2 are cut by this plane. Let e_1 and e_2 be the values for which $e_1 = E(A_1, p_3, n_1)$ and $e_2 = E(A_2, p_3, n_1)$. Given that $\Theta(CMP_1(P)) > \Theta(CMP_2(P))$, this implies that $e_1 > e_2$. Let β_1 be the point (e_1, p_3, n_1) and β_2 be the point (e_2, p_3, n_1), then Figure 4.6 illustrates this situation.

To complete the proof, consider a third cross-sectional plane such that N is always fixed at n_1. Figure 4.7 shows the relationship between all three cross-sectional planes. The N = n_1 plane will cut the efficiency surfaces of algorithm A_1 and algorithm A_2. While the exact curves for the intersection between the N = n_1 plane and the efficiency surfaces for algorithm

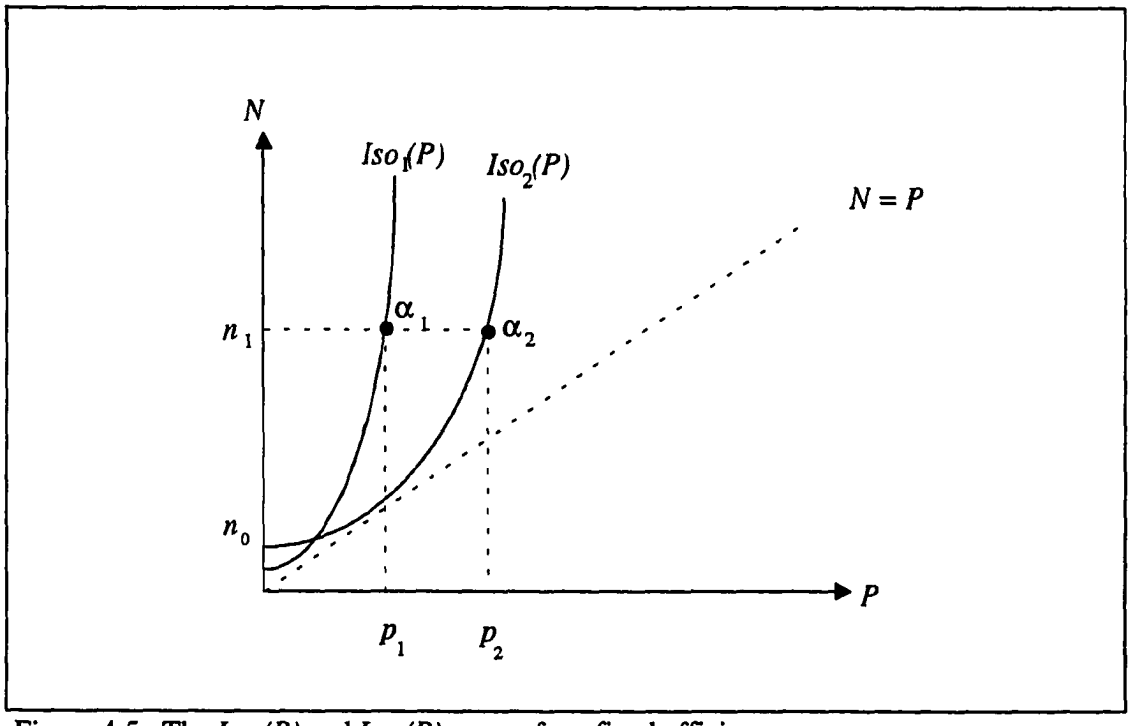

Figure 4.5. The $Iso_1(P)$ and $Iso_2(P)$ curves for a fixed efficiency = e_3.

A_1 and algorithm A_2 might not be known, the relative positioning of points α_1, α_2, β_1, and β_2 on each curve is known. Previously, we have shown that $p_1 < p_2$ and $p_3 > p_2$. This implies that $p_1 < p_2 < p_3$. Since $e_1 = E(A_1, p_3, n_1)$, $e_3 = E(A_1, p_1, n_1)$, and $p_1 < p_3$, this implies that $e_1 < e_3$ because efficiency decreases if you solve the same size problem on a larger number of processor using the same algorithm. Previously, we have shown that $e_1 > e_2$. Therefore, $e_2 < e_1 < e_3$. From $p_1 < p_2 < p_3$ and $e_2 < e_1 < e_3$ we conclude that

$$\left|\frac{e_1-e_3}{p_1-p_3}\right| < \left|\frac{e_2-e_3}{p_2-p_3}\right|. \tag{4.12}$$

Figure 4.8 shows the relative positioning of these points in the N = n_1 plane. Thus, an increase in the number of processors for both algorithms to p_3,while keeping the problem size

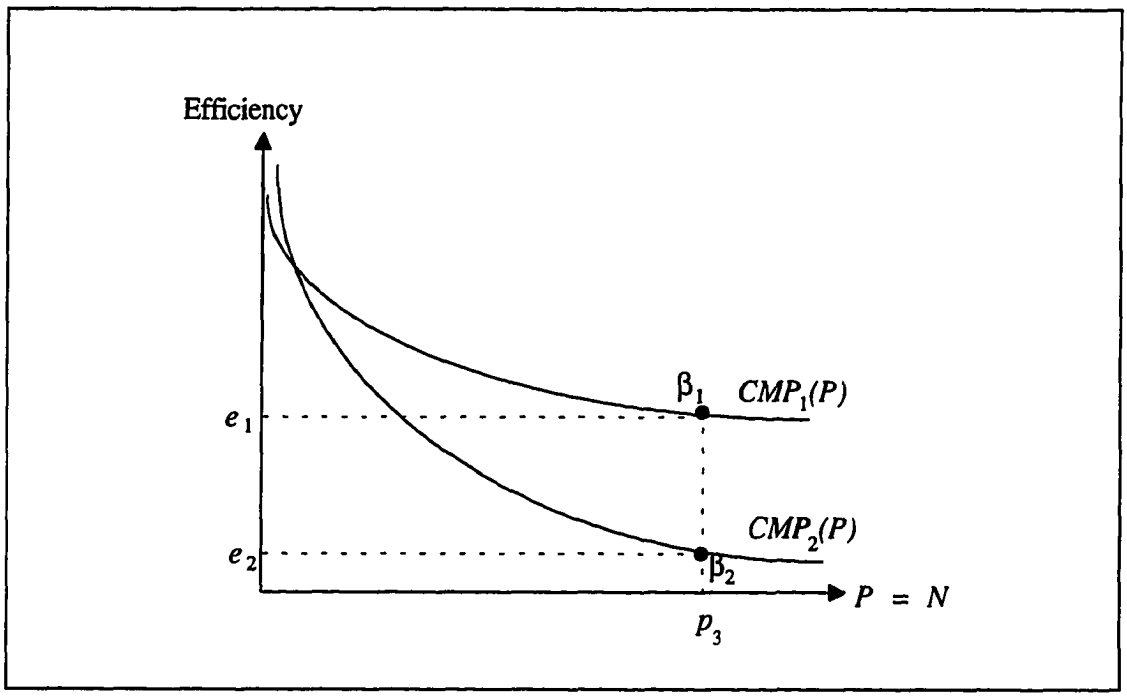

Figure 4.6. The $CMP_1(P)$ and $CMP_2(P)$ curves in the $P = N$ plane.

fixed at n_1, will cause a greater drop in efficiency for algorithm A_2 than for A_1. This completes the proof of theorem 4.1.

Theorem 4.2 is the fixed-machine size scaling analog of Theorem 4.1. It applies to the situation when two different algorithms being compared have conflicting scaling information from the isoefficiency and CMP scalability functions, and you are interested in how the two algorithms will compare under fixed-machine size scaling. Theorem 4.2 is as follows:

Theorem 4.2. Let E(A, P, N) be the efficiency when algorithm A is executed on P processors for a problem size of N. Let e_1 and e_2 be the efficiencies that result from

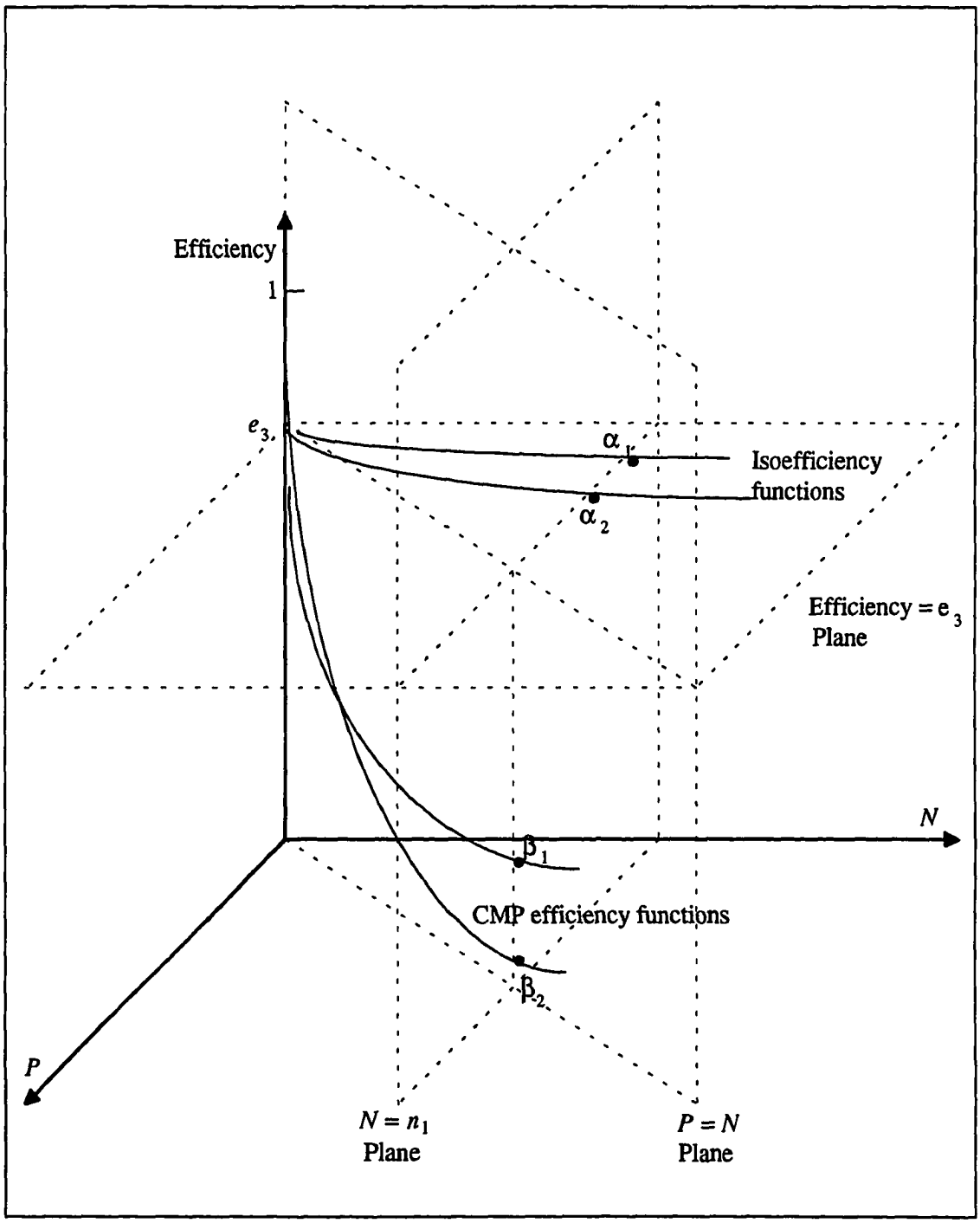

Figure 4.7. The relationship between the constant efficiency = e_3 plane, the $P = N$ plane, and the fixed-problem size = n_1 plane.

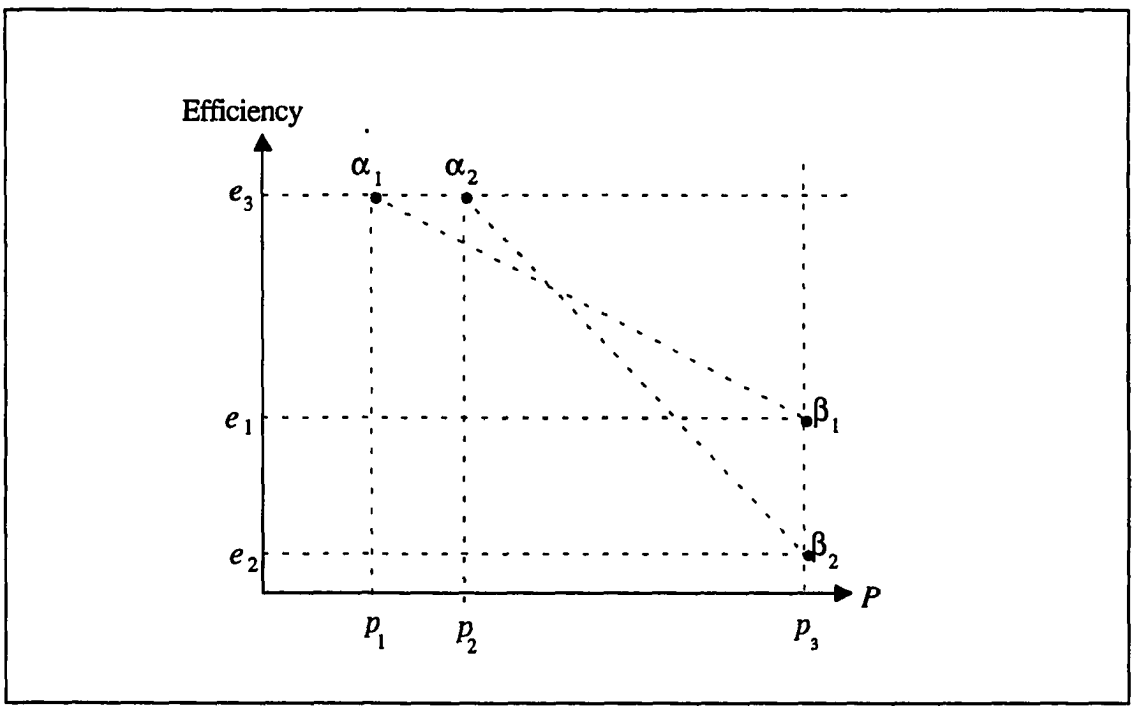

Figure 4.8. α_1 and β_1 are known points of algorithm A_1 while α_2 and β_2 are known points of algorithm A_2.

executing a problem size of n_3 on p_1 processors for algorithms A_1 and A_2 respectively, i.e., $E(A_1, p_1, n_3) = e_1$ and $E(A_2, p_1, n_3) = e_2$. Let the isoefficiency functions $Iso_1(P)$ and $Iso_2(P)$ for algorithms A_1 and A_2 be such that $\Theta\ (Iso_1(P)) > \Theta\ (Iso_2(P))$. Let the CMP-scalability functions $CMP_1(P)$ and $CMP_2(P)$ for algorithms A_1 and A_2 be such that $\Theta\ (CMP_1(P)) > \Theta\ (CMP_2(P))$. Then, an increase in the problem size for algorithms A_1 to n_1 and algorithm A_2 to n_2 such that $E(A_1, p_1, n_1) = E(A_2, p_1, n_2) = e_3$,, while keeping the number of processors fixed, will cause a greater increase in efficiency for algorithm A_2 will be greater than for algorithm A_1.

Proof: The proof for Theorem 4.2 is similar to the proof for Theorem 4.1, since it

involves three cross-sectional planes of the efficiency surfaces for the algorithms. Here, the cross-sectional planes are the fixed-efficiency plane where efficiency = e_3, the $\beta P = N$ plane where $\beta = n_3 / p_1$, and the constant-processor plane where $P = p_1$. These three planes are shown in Figure 4.9.

First, consider the fixed-efficiency plane where efficiency = e_3. The $Iso_1(P)$ and $Iso_2(P)$ functions describe how the efficiency surfaces of algorithm A_1 and algorithm A_2 are cut by this plane. Give that $\Theta\ (Iso_1(P)) > \Theta\ (Iso_2(P))$, this implies $n_1 > n_2$. Let α_1 be the point (e_3 , p_1 , n_1) and α_2 be the point (e_3 , p_1 , n_2) as illustrated in Figure 4.9.

Now, consider the CMP cross-section plane, where $\beta P = N$, where $\beta = n_3 / p_1$. The $CMP_1(P)$ and $CMP_2(P)$ functions describe how the efficiency surfaces of algorithm A_1 and algorithm A_2 are cut by this plane. Let e_1 and e_2 be the values for which $e_1 = E(A_1, p_1, n_3)$ and $e_2 = E(A_2, p_1, n_3)$. Given that $\Theta\ (CMP_1(P)) > \Theta\ (CMP_2(P))$, this implies that $e_1 > e_2$. Let β_1 be the point (e_1 , p_1 , n_3) and β_2 be the point (e_2 , p_1, n_3), as shown in Figure 4.9.

To complete the proof, consider a third cross-sectional plane such that P is always fixed at p_1. Previously, we have shown that $n_1 > n_2$ and $n_2 > n_3$. This implies that $n_1 > n_2 > n_3$. Since $e_1 = E(A_1, p_1, n_3)$ and $e_3 = E(A_1, p_1, n_1)$, and $n_1 > n_3$, this implies that $e_1 < e_3$ because efficiency increases if you solve a larger size problem on the same number of processor using the same algorithm. Previously, we have shown that $e_1 > e_2$. Therefore, $e_2 < e_1 < e_3$. From $n_1 > n_2 > n_3$ and $e_2 < e_1 < e_3$ we conclude that

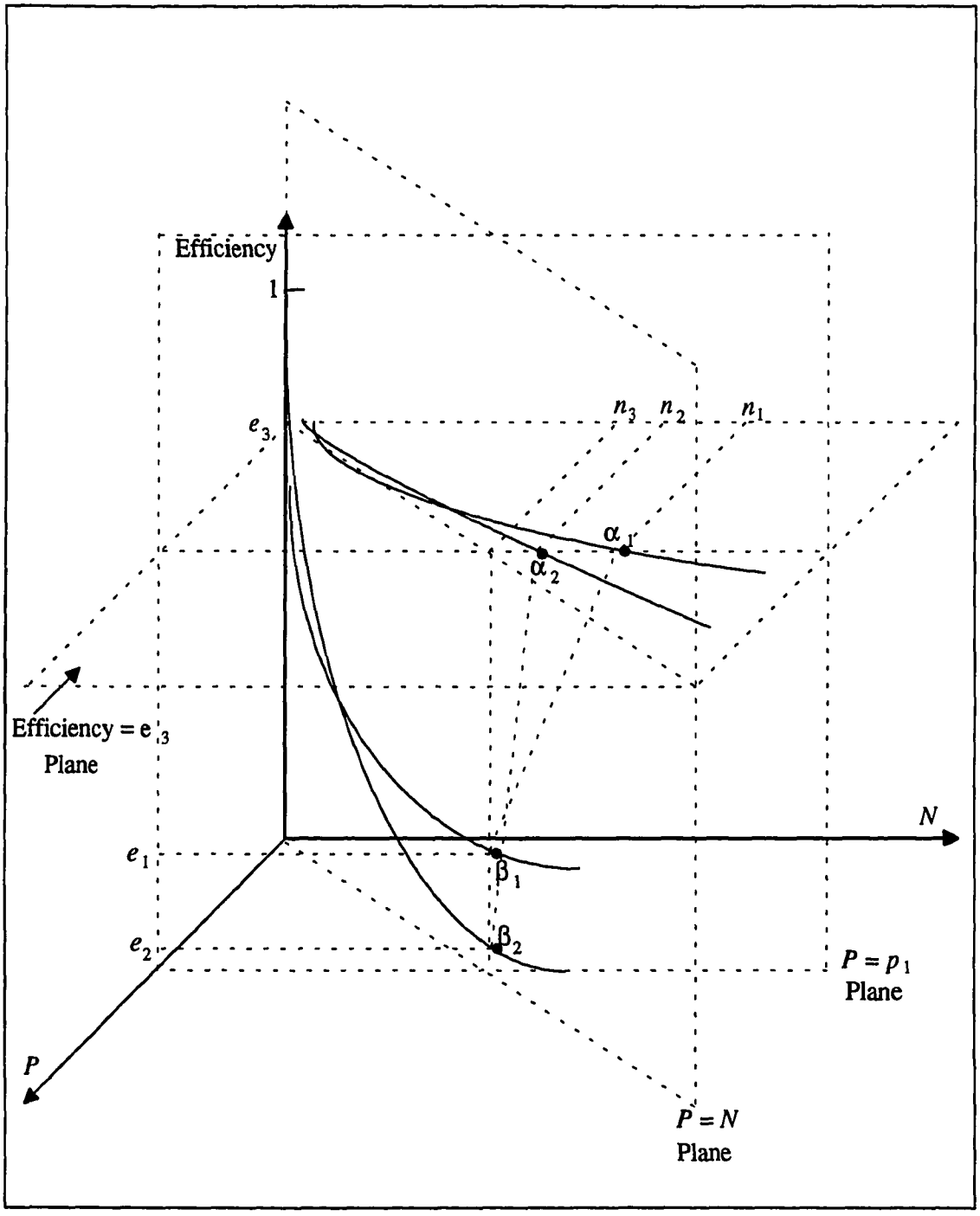

Figure 4.9. The relationship between the constant efficiency = e_3 plane, the $P = N$ plane, and the fixed-problem size = p_1 plane.

$$\left|\frac{e_3-e_1}{n_3-n_1}\right| < \left|\frac{e_3-e_2}{n_3-n_2}\right|. \tag{4.13}$$

Figure 4.9 shows the relative positioning of points α_1, α_2, β_1, and β_2 on each curves. Thus, an increase in the problem size for both algorithms A_1 and A_2 to achieve the same increased efficiency e_3,, while keeping the number of processors fixed, will cause a greater increase in efficiency for algorithm A_2 will be greater than for algorithm A_1. This completes the proof of theorem 4.2.

Theorems 4.1 and 4.2 are valid only in the asymptotic range, i.e., the problem sizes and number of processors is large enough so that $Iso_1(P) > Iso_2(P)$ *and* $CMP_1(P) > CMP_2(P)$ for all the Ps.

4.5. Classification of Non-Fixed-Time Scalable Algorithms

In section 4.2 it was demonstrated that the matrix multiplication, Gauss-Jordan elimination, and Fast Fourier Transform algorithms on mesh and hypercube topologies are not fixed-time scalable. By examining why these algorithms are not fixed-time scalable, general principles can be extracted to determine when an algorithm is not fixed-time scalable. To be fair, it should be pointed out that all algorithms are not scalable under fixed-problem size scaling. Additionally, there are practical limitations to constant-efficiency scaling since the amount of memory contained per processor cannot grow infinitely.

For matrix multiplication, nearest-neighbor communication is performed for each of the $\sqrt{P}$ steps of the algorithm. Since the number of steps grows as P increases and each step represents a submatrix multiplication, the overall parallel execution time will increase unless the local problem size is decreased. However, there is a limit to the amount that the local

problem size can shrink, so matrix multiplication is asymptotically not fixed-time scalable. Similarly, the FFT algorithm on the hypercube topology is not asymptotically fixed-time scalable since the number of communication stages is $\log_2\sqrt{P}$.

Extracting the commonality from both of these cases the generalized Theorem 4.3 can be stated as:

Theorem 4.3: Any parallel algorithm containing a section of code where the number of times it execute is a monotonically increasing function of P is not fixed-time scalable.

The proof of Theorem 4.3 is as follows. Let A be an algorithm containing a section of code *s* where the number of times it executes is a monotonically increasing function *f(P)* as *P* increases; let t_s be the time to execute *s* once; and let *t* be the fixed-time scaling constraint. Since the total time to execute this section of code is t_s*f(P)* and *f(P)* is a monotonically increasing function, t_s*f(P)* will exceed *t* for large enough values of *P*. Therefore, algorithm A is not fixed-time scalable.

GJE and FFT are not fixed-timed time scalable for a different reason. Both algorithms communicate a message between processors whose distance apart is a function of the number of processors. For example, GJE communicates the pivot row and row multipliers along a row or column of the mesh (or its embedding in a hypercube). As the number of processors grows, the time to perform such a communication increases. Therefore, for a large enough number of processors, the time to perform such a communication will exceed any fixed time constraint.

Extracting the commonality from both of these cases the generalized Theorem 4.4 can

be stated as:

Theorem 4.4: Any parallel algorithm where information flows between two processors whose distance is a monotonically increasing function of P is not fixed-time scalable. The flow of information can occur in one step or in multiple steps of the algorithm.

The proof of Theorem 4.4 is trival. Let A be an algorithm where information flows between two processors p_a and p_b whose distance is a monotonically increasing function $f(P)$ as P is increased, and let t be the fixed-time scaling constraint. Regardless of the routing method or the number of communication operations required for the information to flow between p_a and p_b, the total per-hop time required is $f(P)*t_h$, where t_h is an individual hop time of a communication. Since t and t_h are constants and $f(P)$ is a monotonically increasing function, $f(P)*t_h$ will exceed t for large enough values of P. Therefore, algorithm A is not fixed-time scalable.

Unfortunately, Theorem 4.3 and Theorem 4.3 apply to most parallel algorithms on conventional topologies, so asymptotic fixed-time scalability is not generally achievable. It is interesting to consider the types of algorithms and topologies that are fixed-time scalable. A completely interconnected topology is guaranteed to be fixed-time scalable as far as Theorem 4.4 is concerned for any algorithm. Clearly such an interconnection topology is not scalable with todays technology.

CHAPTER 5. MASPAR IMPLEMENTATIONS

In this chapter the implementation details of the parallel algorithms on the MasPas architecture are examined. The discussion starts with an overview of the MasPar architecture, then proceeds to the implementation details including execution-time formulas for the MM, GJE, and FFT algorithms on the MasPar architecture. Experimental verification of the execution time formulas is provided.

5.1. MasPar Architecture

The MasPar architecture is a SIMD architecture consisting of the Array Control Unit (ACU) and a two-dimensional array of processing elements (PEs). Each PE has a load-store architecture with forty 32-bit registers and a local memory.

Two communication subsystems, the *XNET* and the *router*, allow information to flow within the PE array. The XNET subsystem connects each PE to all eight of its neighbors (N, S, E, W, NE, NW, SE, and SW). The XNET connections toroidally-wrap at the edges of the PE array. XNET communication allows all of the active PEs to simultaneously communicate with PEs that are a fixed distance away in one of the eight directions. The router subsystem allows each PE to communicate in a nonuniform pattern with any other PE, but the router can be significantly slower than the XNET depending on the exact communication pattern. Both of these forms of communication are "blocking" in nature, i.e., other useful computation cannot be performed while communication is taking place.

One limitation of the MasPar architecture is that the size of the data block that can be communicated is fine grained. The maximum size of a data block is 64-bits. Additionally,

the data to be communicated must be loaded into a register before it can be communicated. Thus, for a block of data to be transferred from the local memory of one PE to the local memory of another PE, it must be loaded an element at a time into the sender's PE-register, transmitted an element at a time to the receiver's PE-register, and stored into the receiver's memory from its register.

Both pipelined and nonpipelined XNET communication instructions are available. The pipelined (xnetp[d] and xnetc[d]) instructions are used to communicate between two processors if the intermediate processors are not communicating. At each cycle, one bit of information is pipelined through the intermediate processors, so the number of cycles is proportional to (size of the communication)+(distance communicated). The xnetc[d] instruction copies the communicated value to all intermediate processors, while the faster xnetp[d] instruction does not. If the intermediate processors are also communicating, the nonpipelined xnet[d] instruction must be used. This involves a store-and-forwarding of the communicated message between adjacent processors so the number of cycles is proportional to (size of the communication)*(distance communicated). Table 5.1 summarizes the MasPar

Table 5.1. XNET communication costs of 32-bit messages on the MasPar MP-1 and MP-2.

Routine Scheme	XNET instruction (distance of d hops)	MP-1 Cycles	MP-2 Cycles
Pipelined	xnetc[d]	84 + d	48 + d
	xnetp[d]	58 + d	48 + d
Non-pipelined	xnet[d], d = 1	43	40
	xnet[d], d > 1	19 + 35d	13 + 33d

communication instructions for 32-bit messages traveling a distance of d hops.

The MasPar PE architecture does not include cache memory but it allows overlapping of the memory accesses with computation and communication. Four load and/or store operations may be queued while other processing is being performed. This feature of the MasPar architecture can be used to substantially reduce the memory access penalty. Unfortunately, the MasPar MPL (extended C) compiler does not automatically take the best advantage of the memory overlap so the programmer must arrange the load/store instructions in their code to further optimize the memory overlap. This technique for improving the memory overlap in called *software pipelining* [21].

Figure 5.1 demonstrates the basic idea of software pipelining for the matrix multiplication C = A x B, where A, B, and C are N x N matrices. In the unoptimized code (Figure 5.1a) the updating of the c register is stalled until the previous load operations can be performed. The software pipelined code (Figure 5.1b) starts prefetching the operands needed for the updating of the c register during the $(k+1)^{th}$ iteration of the inner loop during the k^{th} loop iteration. Thus, the loading of the A and B elements needed for $(k+1)^{th}$ iteration are overlapped with the addition and multiplication operations needed to update the c register during the k^{th} iteration.

Two MasPar machines, a 16K processor MP-1 and a 4K processor MP-2, were used to run the implemented algorithms. Both MP-1 and MP-2 run at a clock speed of 12.5 MHz. The instruction set is the same for both machines, but the MP-2 processors are faster. Table 5.2 shows the number of clock cycles for critical instructions on the PEs of both machines.

```
// Before Software pipelining
register a, b, c;

for i = 0 to N-1 do
   for j = 0 to N-1 do
      c = 0;
      for k = 0 to N-1 do
         a = A[i, k];
         b = B[k, j];
         c += a * b;
      end for
   end for
end for
```

(a)

```
// After Software pipelining
register a0, a1, b0, b1, c;

 for i = 0 to N-1 do
    for j = 0 to N-1 do
       c  = 0;
       a0 = A[i, 0];
       b0 = B[0, j];
       for k = 0 to N-1 do
          a1 = A[i, k+1];
          b1 = B[k+1, j];
          c += a0 * b0;
          a0 = a1;
          b0 = b1;
       end for
       c += a0 * b0;
       C[i, j] = c;
    end for
end for
```

(b)

Figure 5.1. Example of software pipelining: (a) Non-software pipelined matrix multiplication, and (b) Software pipelined matrix multiplication.

These are actually measured cycle times using the data parallel unit (DPU) timer. The PEs of the MP-2 are roughly four to five times faster at performing arithmetic operations. The local memory is also faster on MP-2 processors. The communication hardware is identical on the MP-1 and MP-2.

The actual MP-1 machine used had 16K processors arranged in a 128x128 array, while the MP-2 machine used a 64x64 PE array for a total of 4K processors. Due to the larger memory size per PE on the MasPar 2, both machines have the same total amount of PE memory (256 M). Software options enable only a portion of the processors to be used when

Table 5.2 Cycles per operation on the MasPar MP-1.

	Operation	MP-1 Cycles	MP-2 Cycles
T_{LD}	Load (not overlapped)	85	40
T_{ST}	Store (not overlapped)	74	35
T_{M}	Floating Point Multiplication	225	41
T_{A}	Floating Point Addition	127	26
T_{D}	Floating Point Division	325	75
T_{NEG}	Floating Point Negation	36	10
$T_{COMPARE}$	Floating Point Comparison	84	33
Tw	Initial Twiddle Factor Calculation	9540	2845

executing a program. Specifically, 32 x 32 and 64 x 64 processors arrays were useful in studying the CMP scaling of the algorithms.

5.2. Algorithm Implementations on the MasPar

5.2.1. Matrix Multiplication

The matrix multiplication implementation on the MasPar did not vary from the algorithm outlined in Figure 2.1. Optimizations performed on the MasPar implementation were software pipelining and loop unrolling. Software pipelining was performed when communicating the submatrices and performing the local submatrice multiplications. Loop unrolling to a depth of 4 was performed to reduce the loop-associated overhead. Further loop unrolling was not possible due to the limited supply of registers.

5.2.2. Gauss-Jordan Elimination

The basic implementation of GJE on the MasPar did not change from the algorithm outlined in Figure 2.2. However, to understand the resulting execution time formulas

(presented later in this chapter) it is necessary to describe the MasPar specific details of the implementation.

In order to find the best pivot element, the PEs storing the pivot column determined a local maximum for their portion of the pivot column. After this, these PEs performed a parallel-prefix "max" computation to determine the globally maximal pivot element. In addition to communicating the pivot row values, the row number corresponding to the pivot element was communicated. When the global maximal pivot value is determined, it is written to a DPU register for convenient access of all PEs.

In broadcasting the new pivot row only the elements to the right of the pivot element are communicated. To perform this, each PE had a temporary row of storage to receive its part of the broadcast pivot row. The xnetc (XNET copy) instruction was used to broadcast elements of the pivot row along the columns of processors and deposit a copy in each PE's register. The new pivot row element was then transfered to the temporary row used to store the pivot row. After the new pivot row is stored in the temporary row, the row of PEs containing the old pivot row broadcast the old pivot row using xnetp to the row of processors that originally contained the new pivot row. This was done to complete the exchange of old and new pivot rows.

The row multipliers were determined by the column of PEs containing the pivot element. Once this is complete, a whole column of row multipliers was broadcast to the other PEs using the xnetc instruction.

Software pipelining was used whenever possible throughout the implementation.

Additionally, all of the above steps as well as updating the matrix where the loop unrolled to a depth of 2.

5.2.3. Fast Fourier Transform

To achieve high performance of FFT on the MasPar machines several options were considered for 1) the layout of butterfly operations on the PE array, (2) the scheme for supplying necessary twiddle factors, (3) the order of evaluation for butterfly operations, (4) the communication scheme, and (5) the strategy to reduce the memory access penalty. These issues are not orthogonal and involve various tradeoffs. In this section, the important performance optimization techniques will be described that lead to an efficient implementation of FFT on the MasPar architecture.

As discussed before, memory accesses are needed if the data to be communicated are not in registers. The first optimization minimizes such memory accesses. This optimization technique comes into the picture when the length of the input sequence is at least four times the number of processors. In a straight *stage-at-a-time evaluation* each PE has multiple butterfly operations to be done for each stage. As the problem size grows, it is not possible to keep the operands for these multiple butterfly operations in registers. The stage-at-a-time evaluation requires accessing an operand of a butterfly operation twice, once for computation and the second time for communication.

These memory accesses are eliminated (except for the initial load at the starting stage and the store at the final stage for each data item) by an optimization technique which uses a different order of evaluation than the stage-at-a-time evaluation. This optimization exploits

the divide-and-conquer nature of FFT. Beyond the in-memory stages, multiple butterfly operations at a processor in fact belong to separate sub-FFTs. The optimized order of evaluation carries each sub-FFT to completion instead of the stage-at-a-time evaluation. Data elements and twiddle factors for a sub-FFT are loaded into registers in the first communication-stage of the FFT. Butterfly operations for subsequent stages use registers for intermediate results with the final stage of the FFT storing the results to memory. For the 32-point FFT example with a 2x2 PE array (Figure 3.5a), data elements and twiddle factors for butterflies $B_{0,3}$ to $B_{3,3}$ (each on a different PE) are loaded into registers at stage 3 (the first stage after the in-memory stages). Butterflies $B_{0,3}$ to $B_{3,3}$ are computed at four different processors for stage 3 saving the results in registers for XNET communication of data elements to butterflies $B_{0,4}$ to $B_{3,4}$ of the next stage. Butterflies $B_{0,4}$ to $B_{3,4}$ are again computed simultaneously for this last stage at four different processors and the results are saved in the memory.

Software pipelining is utilized to reduce the memory-access penalty of the load and store operations which are not eliminated by the above optimization. Here software pipelining is also used when communicating multiple data items. The access of the next data element to be communicated is overlapped with the communication of the current data element.

The final optimization involves the options for supplying the twiddle factors for the butterfly operations. Before stage 0 of the computation, the initial twiddle factors must be calculated using the sine and cosine functions via the equation 3.12. Recalculation of twiddle

factors utilizing equation 3.12 before each stage would be time consuming. One alternative, called "square-of-twiddles", makes use of the modular nature of the twiddle factors to generate a stage's (except the initial stage) twiddle factors from the previous stage's twiddle factors [28]. The square-of-twiddles method squares a butterfly's twiddle factor and selectively negates it if the resulting power is greater than N. This method is substantially faster than calculating the twiddle factors via equation 3.12 since multiplication and negation are much faster than calling the sine and cosine functions. One drawback of the square-of-twiddles method is the accumulation of round-off errors incurred by the repeated squaring of the twiddle factors.

A third alternative for supplying the twiddle factors, which was found to be the best solution on the MasPar architecture, is to use the initial twiddle factors calculated by equation 3.12 and redistribute them for the subsequent stages. This avoids the accuracy problems of the square-of-twiddles method, and it is actually faster. Here the tradeoff is between the time for communicating a twiddle factor versus the time for squaring (and possibly negating) a twiddle factor.

5.3. MasPar Execution-time Formulas for MM, GJE, and FFT

In this section the execution-time formulas for the MM, GJE, and FFT on the MasPar architecture are presented. The execution-time formulas for each algorithm are split into computation time, memory-access time, and communication time. In this chapter the memory-access time formulas are overestimates since they do not account for the overlapping of memory instructions with other computation or communication. The next

chapter corrects this simplification as well as accounts for all other miscellaneous overhead instructions. Additionally, the next chapter experimentally verifies the computation and communication formulas.

5.3.1. Matrix Multiplication

The execution-time formulas for performing an N x N matrix multiplication on a $\sqrt{P} \times \sqrt{P}$ MasPar architecture are

$$T_{COMP}^{MM} = \sqrt{P}\left[\left(\frac{N}{\sqrt{P}}\right)^3 (T_A + T_M)\right] \tag{5.1}$$

$$T_{MEM}^{MM} = 2\sqrt{P}\left[\left(\frac{N}{\sqrt{P}}\right)^2 (3T_{ST} + 2T_{LD}) + \left(\frac{N}{\sqrt{P}}\right)^3 (2T_{LD})\right] \tag{5.2}$$

$$T_{COMM}^{MM} = \sqrt{P}\left(\frac{N}{\sqrt{P}}\right)^2 T_{XNN} \tag{5.3}$$

where T_A is the time to perform an addition, T_M is the time to perform a multiplication, T_{ST} is the time to perform a store-memory access, T_{LD} is the time to perform a load-memory access, and T_{XNN} is the time to perform a nearest-neighbor XNET communication.

5.3.2. Gauss-Jordan Elimination

The MasPar execution-time formulas for solving a linear system of equations Ax = b, where A is an N x N coefficient matrix are

$$T_{COMP}^{GJE} = N[\frac{N}{\sqrt{P}}(2T_{COMPARE} + T_{NEG} + 2T_M + T_A) + \log_2\left(\sqrt{P}\right)T_{COMPARE} + \frac{N^2}{2P}(T_M + T_A) + T_D] + \frac{N}{\sqrt{P}}T_M \tag{5.4}$$

$$T_{MEM}^{GJE} = N[\frac{N}{2\sqrt{P}}(8T_{LD} + 7T_{ST}) + T_{LD} + T_{ST} + \frac{N^2}{2P}(2T_{LD} + T_{ST})] + \frac{N}{\sqrt{P}}(2T_{LD} + T_{ST}) \tag{5.5}$$

$$T_{COMM}^{GJE} = N\left[2T_{XPStart}\log_2\sqrt{P} + 2T_{XP}\sqrt{P} + \frac{N}{2\sqrt{P}}T_{XPStart} + \frac{N}{8}T_{XP} + \frac{3N}{2\sqrt{P}}T_{XCStart} + \frac{3}{2}T_{XC}N\right] \tag{5.6}$$

where $T_{COMPARE}$ is the time to perform a floating-point comparison, T_{NEG} *is* the time to perform a negation, T_D is the time to perform a division, $T_{XPStart}$ is the startup time for an xnetp instruction, T_{XP} is the per-hop transmission time for an xnetp instruction, $T_{XCStart}$ is the startup time for an xnetc instruction, and T_{XC} is the per-hop time for an xnetc instruction.

5.3.3. Fast Fourier Transform, FFT

For an N-point FFT on a $\sqrt{P}$ x $\sqrt{P}$ MasPar architecture, the execution-time formulas are

$$T_{COMP}^{FFT} = \frac{N}{2P}[T_W + 8(T_M + T_A)\log_2(N)] \tag{5.7}$$

$$T_{MEM}^{FFT} = \frac{N}{2P}[6T_{LD} + 6T_{ST} + 6(T_{LD} + T_{ST})\log_2(\frac{N}{P})] \tag{5.8}$$

$$T_{COMM}^{FFT} = \frac{2N}{P}T_{XStart}\log_2 P + 4T_X\sqrt{P}N \tag{5.9}$$

where T_W is the time to calculate an initial twiddle factor by (2.12), T_{XStart} is the time to start a nonpipelined xnet instruction, and T_X is the per-hop time to for the xnet instruction.

CHAPTER 6. SCALABILITY EXPERIMENTS

A 16K-processor MasPar MP-1 machine and a 4K-processor MasPar MP-2 machine were utilized to perform scalability experiments using the matrix multiplication, Gauss-Jordan elimination, and Fast Fourier Transform programs. To study the effects of scaling the number of processors and size of the problems were varied. The problem sizes used ranged from several elements per processor to the largest problems sizes solvable on the machines. Section 6.1 reports the execution-time measurements performed for each algorithm. From these timings, the effect of fixed-problem size scaling, and memory-bounded scaling could be studied directly, but are limited by the number of processors available. To study scalability for a larger number of processors and to study fixed-time scaling, comprehensive execution-time models for the algorithms were developed. Section 6.2 describes how the execution-time models of the three algorithms were developed and verified.

6.1. Execution-Time Measurements

A 16K-processor MasPar MP-1 machine and a 4K-processor MasPar MP-2 machine were utilized to perform scalability experiments using the matrix multiplication, Gauss-Jordan elimination, and Fast Fourier Transform programs shown in Appendix B. Each of the programs were compiled using full compiler optimization (-Omax). Unfortunately, the mpl (MasPar's extended C dialect) compiler did not make the best use of PE registers, so the resulting assembly language code was hand optimized. To study the effects of scaling, the number of processors was varied by using a MasPar compiler option to run these programs on processor arrays of 32 x 32, 64 x 64, and 128 x 128 on the MP-1; and processor arrays of

32 x 32 and 64 x 64 on the MP-2.

Tables A.1 - A.15 in Appendix A show the execution-time measurements, speedup, and overall machine efficiency of matrix multiplication, Gauss-Jordan Elimination, and Fast Fourier Transform, respectively on the MasPar MP-1 and MP-2. The speedup and efficiency were calculated using equations 6.1, 6.2, and 6.3 to estimate the execution-time of the parallel algorithm on a single processor.

$$T_1^{MM}(N) = N^3(T_M + T_A + 2T_{LD}) + N^2(T_{ST}) \qquad 6.1$$

$$T_1^{GJE}(N) = N^3/2[T_M + T_A + 2T_{LD} + T_{ST}] + N^2\left[T_{CMP} + \tfrac{7}{2}T_{LD} + \tfrac{3}{2}T_{ST} + T_D\right] \qquad 6.2$$

$$T_1^{FFT} = 4(T_M + T_A)N\log_2 N + NT_\omega/2 \qquad 6.3$$

where T_{LD} and T_{ST} are adjusted to account for the memory overlap achieved by the corresponding parallel program as discribed in section 6.2.

The observed CMP-scaling for MM, GJE, and FFT for a constant-memory-per-processor, β, equal to 1024 elements is shown in Figure 6.1. The expected growth of the CMP speedup from the asymptotic CMP-scalability analyses are O(P) for MM, O($\sqrt{P}$) for GJE, and O($\sqrt{P}\ \log_2 P$) for the FFT. The near linear CMP speedup observed does not match the observed behavior. The next chapter explains inaccuracy of the CMP-scalability metric for the MasPar MP-1.

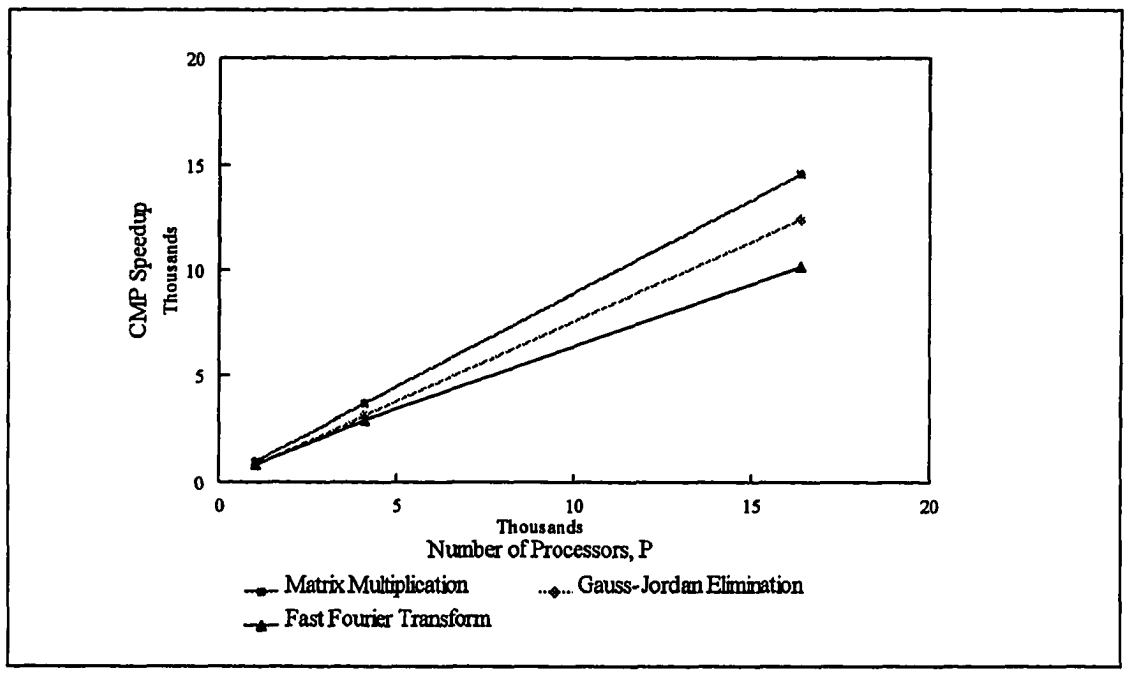

Figure 6.1. CMP speedup on the MasPar MP-1 with $\beta = 1024$ for all algorithms.

6.2. Development of Execution-Time Models

The execution-time models for each program were split into four parts: computation time, communication time, memory-access time, and miscellaneous-overhead time. The computation and communication time formulas developed in chapter 3 were verified, as described below, to be accurate. Refinements to the memory-accesses formulas developed in chapter 3 are provided, since the MasPar processors allow overlapping of memory-accesses with computation or communication. Additionally, miscellaneous overheads including loop-control overhead, register-to-register moves, and array index-pointer manipulations are incorporated into the model.

First, consider the refinement to account for the miscellaneous overheads. The

Table 6.1. Miscellaneous-overhead constants for the MasPar MP-1.

Program	α_1	α_2	α_3	α_4	α_5
Matrix Multiplication	1.21e-4	9.02e-8	3.18e-6	7.18e-6	6.72e-7
Gauss-Jordan Elimination	3.39e-3	1.75e-5	4.35e-5	3.43e-5	3.19e-7
Fast Fourier Transform	1.10e-4	2.51e-5	1.57e-5	3.07e-5	—

Table 6.2. Miscellaneous-overhead constants for the MasPar MP-2.

Program	α_1	α_2	α_3	α_4	α_5
Matrix Multiplication	1.03e-4	2.97e-8	4.40e-6	4.50e-6	6.75e-7
Gauss-Jordan Elimination	1.50e-3	3.87e-5	2.12e-5	1.88e-5	3.25e-7
Fast Fourier Transform	6.46e-5	9.97e-6	5.51e-6	8.63e-6	—

miscellaneous overheads included loop-control overhead, register-to-register moves, and array index-pointer manipulations. The miscellaneous overhead was experimentally measured by timing the programs after deleting instructions for computation, communication, and memory access from the compiler-generated assembly language code. For a small local problem size per processor, miscellaneous overheads of 8% for MM, 43% for GJE were observed. However, as the local problem size was increased the miscellaneous-overhead times decreased and stabilized to 3 % and 5 % of the total execution time for MM and GJE respectively. These percentages do not vary noticeably with the number of processors. The reason for this is that the majority of execution time occurs within loops that are independent

of the number of processors. The miscellaneous overheads for FFT ranged from 16% to 13% as the processor-array size changed from 32 x 32 to 128 x128. The miscellaneous overheads for FFT are split between loops performing the in-memory stages and communication stages. Since the number of stages of each type varies with the number of processors, the percentage of miscellaneous overheads depends on the processor-array size.

In modeling the miscellaneous-overhead times, formulas were developed for each algorithm based on their looping structures. Temples for the formulas were chosen to be similar to the computation-time analytical formulas of the Chapter 5. The temples for MM, GJE, and FFT are

$$T_{MISC}^{MM} = \alpha_1 + \alpha_2 N + \alpha_3 \sqrt{P} + \alpha_4 \frac{N^2}{\sqrt{P}} + \alpha_5 \frac{N^3}{P} \tag{6.4}$$

$$T_{MISC}^{GJE} = \alpha_1 + \alpha_2 N + \alpha_3 N \log_2 \sqrt{P} + \alpha_4 \frac{N^2}{\sqrt{P}} + \alpha_5 \frac{N^3}{P} \tag{6.5}$$

$$T_{MISC}^{FFT} = \alpha_1 + \alpha_2 \frac{N}{P} + \alpha_3 \frac{N}{P} \log_2\left(\frac{N}{P}\right) + \alpha_4 \frac{N}{P} \log_2 P \tag{6.6}$$

where the α_i's are constants that are experimentally determined using the measured miscellaneous-overhead times. For the MasPar MP-1 and MP-2, these constants are given in Table 6.1 and Table 6.2.

Secondly, a refinement was done to account for the overlapping of memory-access times with other computation. After accounting for the computation, communication, and miscellaneous-overhead portions of the execution time, the remaining time was attributed to memory-accesses. The amount of reduction in the memory-access times depends on the processing being done at individual processors and it stabilizes as the local problem size

Execution Time is the sum of the computation, communication, miscellaneous, and memory times as determined by:

Computation Time: Use the formulas 5.1, 5.4, or 5.7 from chapter 5.

Communication Time: Use the formulas 5.3, 5.6, or 5.9 from chapter 5.

Miscellaneous Time: Use the formulas 6.1, 6.2 or 6.3 and the machine constants from Tables 6.1, or 6.2.

Memory Time: If the size is less than the number of processor available, extrapolate between measured values to determine the overlap; otherwise use the stablized values for the memory overlap with the formulas 5.2, 5.5, or 5.8.

Figure 6.2. Procedure for predicting the execution time.

became sufficiently large. It was observed that at small local problem sizes per processor little overlap occurred. However, as the local problem size is increased, the memory-access times are reduced by factors of 73%, 79%, and 79% for MM, GJE, and FFT, respectively on the MasPar MP-1.

The actual memory-access times were modeled by using the analytical formulas of chapter 5 and by allowing for the reduction in the memory-access time due to operand prefetching. For small problem sizes per processor, the memory-overlap was interpolated between experimentally measured values to determine the reduction in the memory-access time. For local problem sizes larger than were able to be measured, the stabilized memory-access reductions were utilized.

Figure 6.2 summarized the procedure for predicting the execution time of the

Table 6.3. Model results vs. experimental results.

Rank of Matrix Algorithm		N=128	N=256	N=512	N=1024	N=2048	N=3072	N=4096
	$\sqrt{P}$	Percentage Differences						
MM	32	3.5 %	3.5 %	2.1 %	2.2 %	*	*	*
	64	3.5 %	3.5 %	3.5 %	2.2 %	2.2 %	*	*
	128	*	0.6 %	3.5 %	3.7 %	2.2 %	2.3 %	2.2 %
Rank of Matrix		N=256	N=512	N=1024	N=2048	N=3072	N=4096	N=5120
Algor-ithm	$\sqrt{P}$	Percentage Differences						
GJE	32	0.9 %	0.7 %	0.3 %	*	*	*	*
	64	0.6 %	0.5 %	0.6 %	0.3 %	0.1 %	*	*
	128	3.5 %	1.4 %	0.0 %	0.4 %	0.3 %	0.2 %	0.1 %
Number of Elements		2^{16}	2^{18}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}
Algor-ithm	$\sqrt{P}$	Percentage Differences						
FFT	32	0.4 %	0.4 %	0.5 %	*	*	*	*
	64	0.1 %	0.0 %	0.0 %	0.1 %	0.1 %	*	*
	128	0.5 %	0.5 %	0.5 %	0.5 %	0.4 %	0.4 %	0.3 %
* Indicates insufficient memory								

algorithms. Tables 6.3. summarizes the maximum precentage errors for the combined execution-time formulas versus timed code. The results are very accurate with a maximum error of 3.5% for the matrix multiplication algorithm.

CHAPTER 7. ACCURACY OF ASYMPTOTIC SCALABILITY METRICS IN PRACTICE

Isoefficiency and CMP-scalability are asymptotic scalability metrics that result in a function in terms of P. The isoefficiency function describes how the problem size should grow as P is increased so that a constant efficiency can be maintained, and the CMP-scalability function describes how the speedup should increase as P is increased if the memory usage per processor is kept fixed. Both of these scalability metrics focus on the asymptotically important terms and ignore the remaining terms and constants.

The goals of this chapter are (1) to examine the inaccuracies introduced by these simplifications on "non-asymptotic" problem sizes and machine sizes, and (2) to study the effects that varying the machine specific parameters have on these inaccuracies. To study these inaccuracies, the execution-time formulas developed in the previous chapter for Matrix Multiplication, Gauss-Jordan Elimination, and Fast Fourier Transform on the MasPar MP-1 are used to predict the asymptotic behavior for the CMP and isoefficiency-scalability metrics.

7.1. Measuring Asymptotic Inaccuracies

For the CMP scalability metric, the speedup predicted by the asymptotically significant terms along with their corresponding constants are compared with the speedup predicted by the whole execution-time formulas. Specifically, the percentage error is calculated by the formula

$$\frac{(\text{CMP Predicted Speedup}(\beta, P) - \text{Actual Speedup}(P, N))}{\text{Actual Speedup}(P, N)} * 100, \qquad (7.1)$$

where the "CMP Predicted Speedup(P, N)" is calculated by using the asymptotically

significant terms along with their corresponding constants, and the "Actual Speedup(P, N)" is calculated using the execution-time formulas developed in the previous chapter. Since P and N are not independent variables in CMP scalability, i.e., they are related by the formula $\beta = N/P$ where β is a constant, the percentage error in the efficiency is a function of β and P.

One way of viewing the isoefficiency metric is that it predicts the problem size necessary on a machine with P processors inorder to achieve some constant efficiency. So, to measure the inaccuracy of the isoefficiency function, the following formula is used

$$\frac{|\text{Isoefficiency Predicted Problem Size to Achieve Efficiency e - Actual Problem Size}|}{\text{Actual Problem Size}} * 100 \qquad (7.2)$$

where the "Isoefficiency Predicted Problem Size to Achieve Efficiency e" uses the asymptotically important terms and their corresponding constants. This percentage error for the isoefficiency function has two independent variables: the number of processors (P) (or the problem size (N)), and the efficiency constant chosen, e.

7.2. Accuracy of the CMP Scalibility Function

The asymptotic CMP-scalability function proved to be a poor predictor of speedup on the MasPar MP-1 for Gauss-Jordan Elimination and Fast Fourier Transform. Tables 7.1 and 7.2 show the percentage error as defined by equation 7.1 for the Gauss-Jordan Elimination and Fast Fourier Transform algorithms. For these ranges of processors and problem sizes, the CMP-scalability function was not very accurate (< 10 % error) for the FFT until the processor array of 2048 x 2048 was used, and it was never that accurate for the GJE.

Table 7.1. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	14494%	19608%	26856%	37118%	51640%	72183%
64	7264%	9818%	13440%	18570%	25830%	36100%
128	3639%	4914%	6724%	9289%	12918%	18053%
256	1823%	2460%	3364%	4646%	6460%	9028%
512	915%	1232%	1684%	2324%	3231%	4514%
1,024	460%	618%	843%	1163%	1616%	2258%
2,048	232%	310%	423%	582%	809%	1129%
4,096	118%	157%	212%	292%	405%	565%
8,192	61%	80%	107%	147%	203%	283%
16,384	32%	41%	55%	74%	102%	142%

Table 7.2. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	613%	648%	682%	716%	751%	785%
64	332%	350%	367%	385%	402%	419%
128	176%	185%	194%	203%	212%	221%
256	91%	96%	100%	105%	109%	114%
512	45%	47%	50%	52%	54%	57%
1,024	20%	21%	22%	24%	25%	26%
2,048	7%	7%	8%	9%	10%	10%
4,096	-0%	0%	1%	1%	2%	2%
8,192	-4%	-3%	-3%	-3%	-2%	-2%
16,384	-5%	-5%	-5%	-5%	-4%	-4%

Observe the general trends in these tables. First, the CMP-scalability predictions are generally more accurate as the number of processors increased. This is understandable for an asymptotic scalability metric since the significants of the asymptotic terms increase as the number of processors is increased. Secondly, the predictions were less accurate as the problem size increased. Finally, the CMP-scalability predictions are approximately two orders of magnitude better for the Fast Fourier Transformation algorithm than for the Gauss-Jordan Elimination algorithm.

To explain these trends, it is useful to examine the contribution of individual terms in the sequential and parallel execution formulas. For GJE, the terms of the speedup formula is

$$\frac{c_1N^3+c_2N^2}{c_3\frac{N^3}{P}+c_4N^2+c_5\frac{N^2}{\sqrt{P}}+c_6N\sqrt{P}+c_7N\log_2P+c_8N+c_9\frac{N}{\sqrt{P}}} \tag{7.3}$$

and for FFT the terms of the speedup formula is

$$\frac{c_1N+c_2N\log_2N}{c_3\frac{N}{P}\log_2N+c_4\frac{N}{P}\log_2\left(\frac{N}{P}\right)+c_5\frac{N}{P}+c_6\log_2P+c_7\frac{N}{\sqrt{P}}}, \tag{7.4}$$

where the c_i's are machine specific constants. Table 7.3 for the 2K-memory-per-processor GJE problem and Table 7.4 for the 2K-memory-per-processor FFT problem show the contribution of each c_i term to their respective sequential (the numerator) or parallel (the denominator) execution time. The asymptotically important terms are the c_1 and c_4 terms for GJE, and the c_2 and c_7 terms for FFT (shaded in the tables).

The accuracy of the CMP scalability predictions improves as the number of

Table 7.3. Contribution of each term in the Gauss-Jordan Elimination to their respective sequential or parallel execution time under CMP scaling.

Type of c_i Term	Sequential Terms		Parallel Terms						
	Comp.	Comp.	Comp.	Comm.	Comm. & Comp.	Comm.	Comm. & Comp.	Comp.	Comp.
$\sqrt{P}$	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
32	99.8%	0.2%	88.8%	0.5%	10.3%	0.0%	0.2%	0.1%	0.0%
64	99.9%	0.1%	88.3%	1.0%	10.3%	0.0%	0.3%	0.1%	0.0%
128	100.0%	0.0%	87.4%	2.0%	10.2%	0.1%	0.3%	0.1%	0.0%
256	100.0%	0.0%	85.6%	3.9%	10.0%	0.1%	0.3%	0.1%	0.0%
512	100.0%	0.0%	82.3%	7.5%	9.6%	0.2%	0.4%	0.1%	0.0%
1,024	100.0%	0.0%	76.4%	13.9%	8.9%	0.4%	0.4%	0.1%	0.0%
2,048	100.0%	0.0%	66.8%	24.4%	7.8%	0.7%	0.4%	0.1%	0.0%
4,096	100.0%	0.0%	53.4%	39.0%	6.2%	1.1%	0.3%	0.0%	0.0%
8,192	100.0%	0.0%	38.1%	55.6%	4.4%	1.5%	0.2%	0.0%	0.0%
16,384	100.0%	0.0%	24.3%	70.8%	2.8%	1.9%	0.2%	0.0%	0.0%

Table 7.4. Contribution of each term in the Fast Fourier Transform to their respective sequential or parallel execution time under CMP scaling.

Type of c_i Term	Sequential Terms		Parallel Terms				
	Comp.	Comp.	Comp.	Memory	Comp.	Comm.	Comm.
$\sqrt{P}$	c_1	c_2	c_3	c_4	c_5	c_6	c_7
32	13.5%	86.5%	72.7%	3.1%	12.6%	0.0%	11.6%
64	12.4%	87.6%	67.3%	2.6%	10.6%	0.0%	19.5%
128	11.5%	88.5%	58.5%	2.1%	8.5%	0.0%	31.0%
256	10.7%	89.3%	46.6%	1.5%	6.2%	0.0%	45.6%
512	10.0%	90.0%	33.6%	1.0%	4.2%	0.0%	61.2%
1,024	9.4%	90.6%	22.0%	0.6%	2.6%	0.0%	74.8%
2,048	8.9%	91.1%	13.3%	0.4%	1.4%	0.0%	84.9%
4,096	8.4%	91.6%	7.6%	0.2%	0.8%	0.0%	91.4%
8,192	8.0%	92.0%	4.2%	0.1%	0.4%	0.0%	95.3%
16,384	7.6%	92.4%	2.3%	0.1%	0.2%	0.0%	97.5%

processors increased because the communication terms (c_4 term for GJE and c_7 terms for FFT) double each time that $\sqrt{P}$ doubles. Eventually, the communication terms dominates the denominators, but the major computation-terms in the denominator (c_3 term for GJE and c_3 for FFT) are 404 times and 6 times larger initially for GJE and FFT, respectively. This also explains why the CMP-scalability predictions are approximately two orders of magnitude better for the Fast Fourier Transformation algorithm than for the Gauss-Jordan Elimination algorithm. The rise in the CMP scalability error for FFT after it had reached zero is due to the relatively large c_1 term. If Table 7.2 is extended, the error would rise to approximately 6% for all problem sizes and then slowly decline.

The CMP-scalability predictions are less accurate as the problem size increases because the denominators' communication terms grow slower than the computation terms as the problem size is increased. For GJE, the c_3 computation term has a N^3 multiplier which grows faster than the communication term's N^2. Similarly for FFT, the denominator's computation term has a multiplier of $N\log_2 N$, while the communication term has an N multiplier.

7.3. Effects of Varying Machine Parameters on the Accuracy of CMP Scability

From the above analysis, increasing the speed of the communication relative to computation would be expected to degrade the accuracy of the CMP-scalability predictions, since the communication constants (c_4 term for GJE and c_7 for FFT) would become closer in size to the computation constants (c_3 term for GJE and c_3 for FFT). In fact, a linear relationship between communication speedup and CMP-scalability accuracy would be

expected, since the size of the asymptotic communication terms is linearly effected. To show this, ten-fold, fifty-fold, and one-hundred-fold increases in the communication speed are considered. Figures 7.1 and 7.2 summarize these results by showing the percentage error of the CMP-scalability function for memory usage of 1K, 8K, 16K, and 32K per processor on a 128 x 128 processor array. Tables A.22 through A.26 of Appendix A contain all of the data for these experiments. These figures clearly show that increasing the communications speed linearly degrades the accuracy of the CMP-scalability predictions.

By the same token, increases in the computation speed (and memory speed), while leaving the commication speed the same, would be expected to improve the accuracy of the CMP-scalability funcion. Figure 7.3 for the GJE and Figure 7.4 for the FFT show that this is indeed the case for memory usage of 1K, 8K, 16K, and 32K per processor. Tables A.18 through A.21 of Appendix A contain all of the data for these experiments. The improvement in the accuracy of the CMP-scalabitiy predictions are clearly better than linear as the computation speed is increased. All terms of the sequential execution time (the numerator) decrease linearly with the increased speedup of the computation. Additionally, the computation terms of the parallel execution time decrease relative to the asymptotic communication terms (c_4 term for GJE and c_7 for FFT). The combined effect causes a better than linear improvement in the CMP-scalability predictions as the computational speedup is increased.

The MasPar MP-1 processors are relatively slow in comparison to its communication speed. Other commonly available distributed-memory parallel computers have a higher

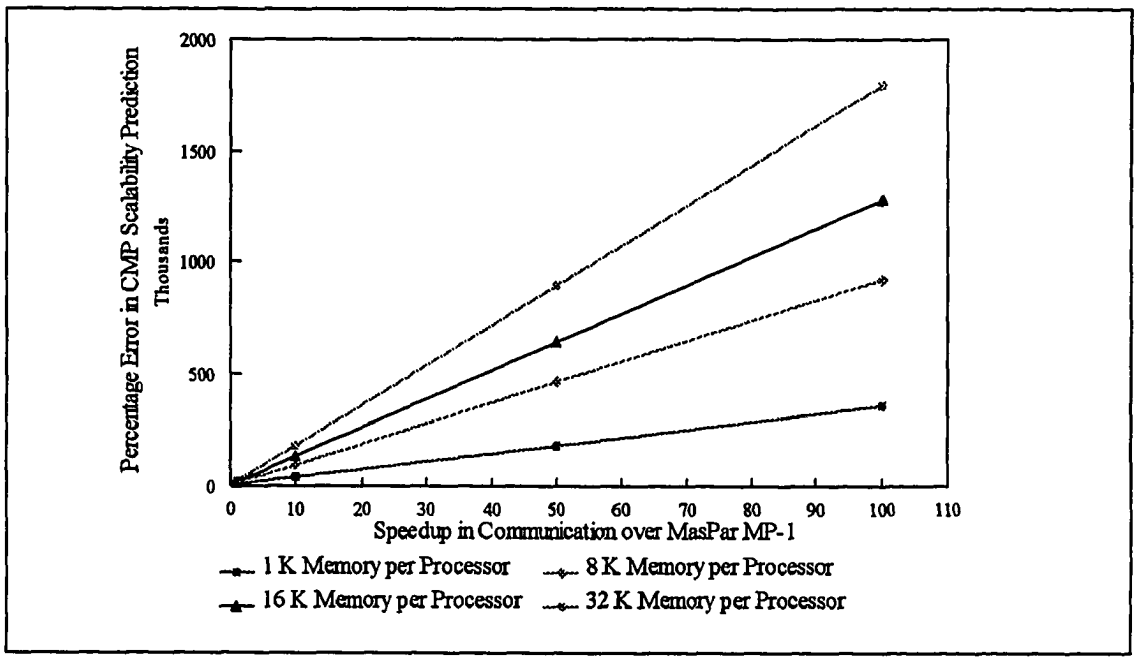

Figure 7.1. Degradation in the CMP scalability accuracy for Gauss-Jordan Elimination as the communication speed is increased for a 128 x 128 processor array.

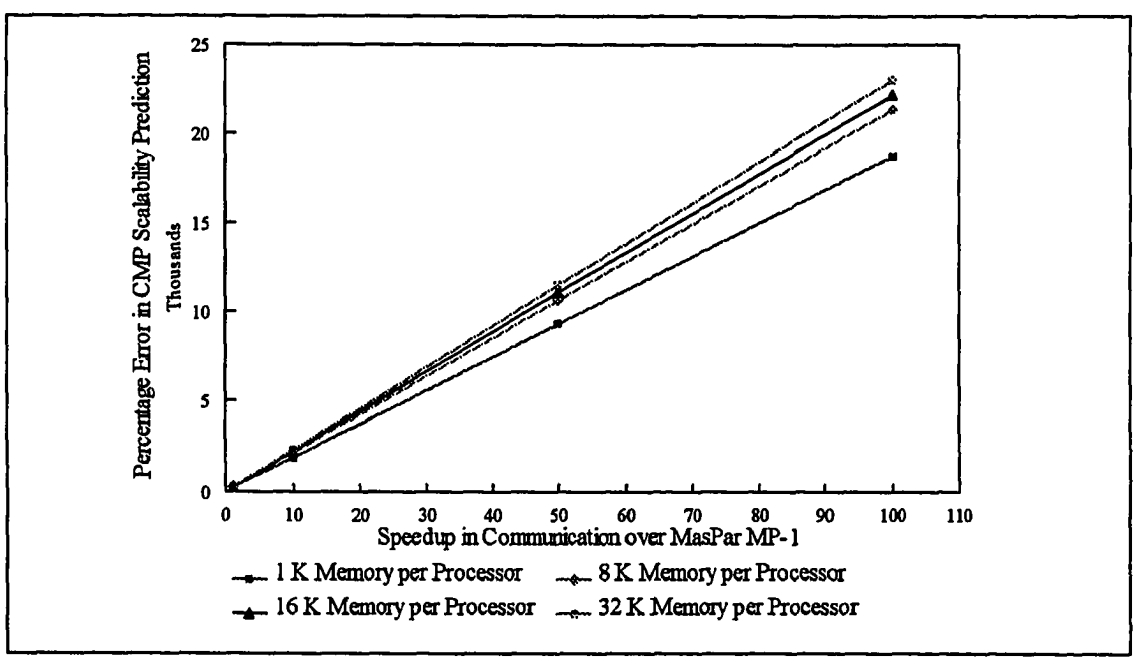

Figure 7.2. Degradation in the CMP scalability accuracy for Fast Fourier Transform as the communication speed is increased for a 128 x 128 processor array.

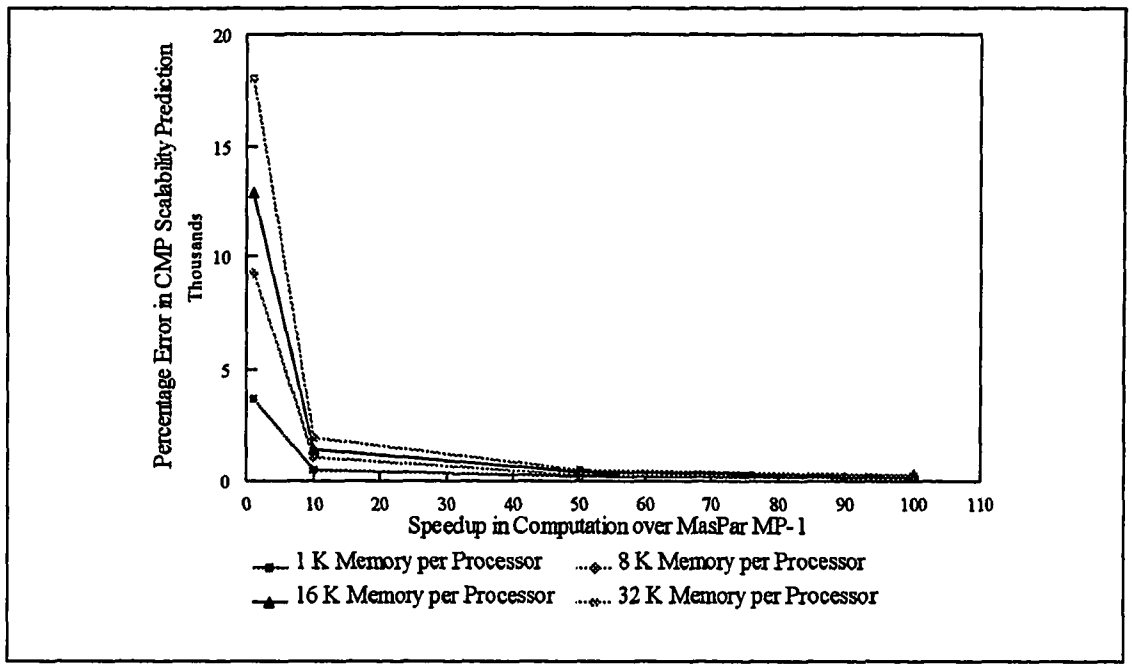

Figure 7.3. Improvement in the CMP scalability accuracy for Gauss-Jordan Elimination computation speed is increased for a 128 x 128 processor array.

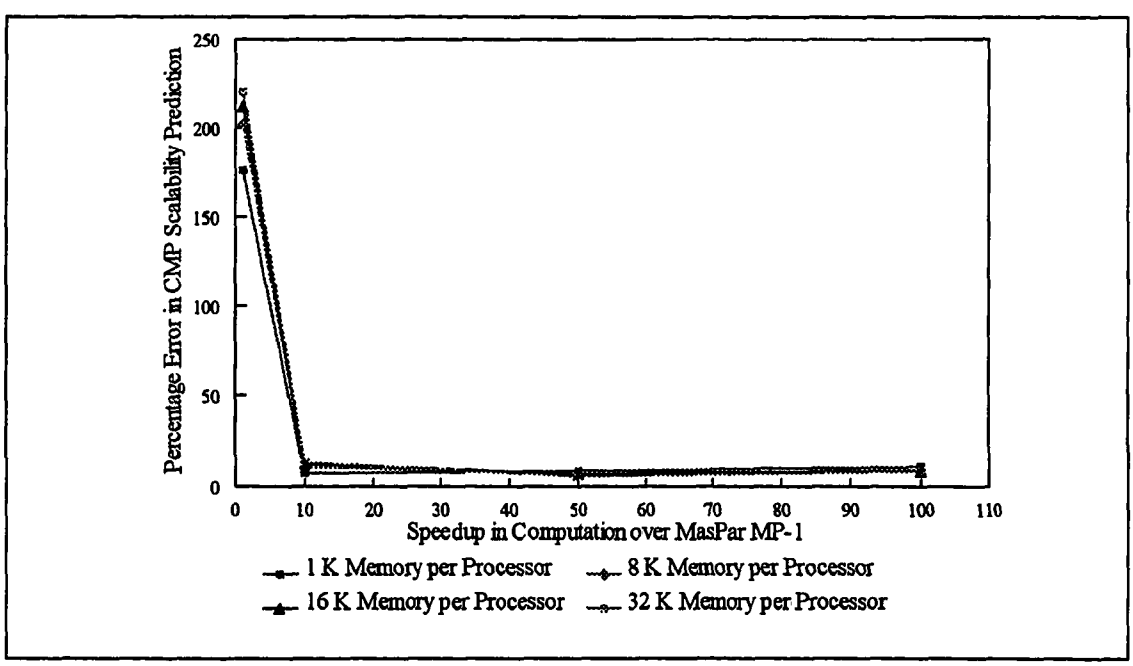

Figure 7.4. Improvement in the CMP scalability accuracy for Fast Fourier Transform as the computation speed is increased for a 128 x 128 processor array.

computation to communication speed ratio. This means that CMP-scalability predictions will be more accurate on these machines.

7.3. Accuracy of the Isoefficiency Function

The accuracy of the isoefficiency scalability function is poor at predicting the problem size necessary to achieve a specified machine efficiency on the MasPar MP-1. The percentage error in the isoefficiency predicted problem sizes for GJE and FFT with a fixed efficiency = 0.8 are shown in Table 7.5 and 7.6, respectively. For GJE, two trends in the percentage errors are apparent: (1) the accuracy improves as processor array increases in size for a constant efficiency, and (2) the accuracy improves as the required efficiency increases for a fixed size processor array. For FFT, these trends are reversed, i.e., (1) the accuracy degrades as the processor array increases in size for a constant efficiency, and (2) the accuracy degrades as the required efficiency increases for a fixed size processor array.

Accuracy errors in the problem size predicted by the isoefficiency function are due to using only the asymptotically important terms when determining the predicted problem size. The predicted problem size for an efficiency of E is

$$T_1(N) = \frac{E}{1-E} T_o(P,N), \tag{7.5}$$

where $T_1(N)$ is the sequential processor execution time and $T_0(P, N)$ is the total overhead time for all processors. Whe predicting the problem size for an efficiency E, the $T_1(N)$ and $T_0(P, N)$ terms are approximated by the asymptotically important sequential and parallel terms.

For GJE, the approximating equation is

Table 7.5. Percentage error of isoefficiency-scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer.

$\sqrt{P}$	Fixed Efficiency			
	0.2	0.4	0.6	0.8
32	97.0%	96.3%	95.8%	95.5%
64	94.4%	93.0%	92.1%	91.5%
128	89.8%	87.2%	85.7%	84.5%
256	82.1%	77.8%	75.1%	73.3%
512	70.4%	64.0%	60.3%	57.9%
1,024	55.0%	47.1%	43.2%	40.7%

Table 7.6. Percentage error of isoefficiency-scalability function for Fast Fourier Transform on a MasPar MP-1 computer.

$\sqrt{P}$	Fixed Efficiency			
	0.2	0.4	0.6	0.8
32	a	a	a	a
64	a	a	a	4010%
128	a	a	a	5400%
256	a	a	1835%	7219%
512	a	1205%	2055%	9466%
1,024	938%	1262%	2303%	14090%

a - Indicates that one element per processor achieved an efficiency that was higher that the corresponding fixed efficiency.

$$c_1 N^3 = \frac{E}{1-E} c_4 P N^2, \tag{7.6}$$

where the c_i's are the same as in equation 7.3. Solving 7.6 for N

$$N = \frac{E}{1-E} \frac{C_4}{C_1} P \tag{7.7}$$

gives the formula used to predict the problem size. The accuracy of the GJE problem-size

Table 7.7. Contribution of each term in the Gauss-Jordan Elimination to their respective sequential or parallel execution time for a constant efficiency of 0.8.

$\sqrt{P}$	Sequential Terms		Parallel Terms						
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
32	99.7%	0.3%	79.7%	0.9%	18.3%	0.0%	0.8%	0.3%	0.0%
64	99.8%	0.2%	79.9%	1.7%	17.3%	0.1%	0.8%	0.2%	0.0%
128	99.9%	0.1%	80.0%	3.1%	15.8%	0.1%	0.8%	0.2%	0.0%
256	100.0%	0.0%	80.0%	5.3%	13.6%	0.2%	0.7%	0.1%	0.0%
512	100.0%	0.0%	80.0%	8.4%	10.7%	0.3%	0.5%	0.1%	0.0%
1,024	100.0%	0.0%	80.0%	11.9%	7.6%	0.3%	0.3%	0.0%	0.0%

prediction depends on how well the c_1 term approximates $T_1(N)$ and how well the c_4 term approximates $T_0(P, N)$. Table 7.7 shows the contribution of all the c_i terms for the efficiency of 0.8 as the number of processors increases. The c_1 term is a very good approximation of $T_1(N)$ contributing 99.7% of the sequential execution time on a 32 x 32 processor array and it only improves as the processor array increases in size. Since the c_4 term should approximate $T_0(P, N)$ and the efficiency is 0.8, the c_4 term should contribute 20% of the total parallel execution time. The constribution of the c_4 term starts out at 0.9% on a 32 x 32 processor array and increases to 11.9% on a 1024 x 1024 processor array. As shown in Table 7.5, the observed percentage error in the problem-size predictions for an efficiency of 0.8 reflect this improvement in the c_4 term's approximation of $T_0(P, N)$.

For FFT, the approximating equation is

$$c_2 N \log_2 N = \frac{E}{1-E} c_7 N \sqrt{P}\,, \tag{7.8}$$

where the c_i's are the same as in equation 7.4. Solving 7.8 for N

Table 7.8. Contribution of each term in the Fast Fourier Transform to their respective sequential or parallel execution time for a constant efficiency of 0.8.

$\sqrt{P}$	Sequential Terms		Parallel Terms				
	c_1	c_2	c_3	c_4	c_5	c_6	c_7
32	a	a	a	a	a	a	a
64	14.7%	84.3%	62.9%	1.8%	12.1%	1.1%	22.1%
128	7.1%	92.9%	68.6%	3.8%	5.8%	0.6%	21.2%
256	3.4%	96.6%	71.2%	4.9%	2.8%	0.4%	20.7%
512	1.7%	98.3%	72.5%	5.5%	1.4%	0.2%	20.4%
1,024	0.8%	99.2%	73.1%	5.8%	0.7%	0.1%	20.2%

a - Indicates that one element per processor gives a higher efficiency than 0.8

$$N = 2^{\frac{E}{1-E}\frac{c_7}{c_2}\sqrt{P}} \tag{7.9}$$

gives the formula used to predict the problem size. The accuracy of the GJE problem-size prediction depends on how well the c_2 term approximates $T_1(N)$ and how well the c_7 term approximates $T_0(P, N)$. Table 7.8 shows the contribution of all the c_i terms for the efficiency of 0.8 as the number of processors increases. The c_2 term is a fair approximation of $T_1(N)$ contributing 84.3% of the sequential execution time on a 32 x 32 processor array and it improves as the processor array increases such that it contributes 99.2% on a 1024 x 1024 processor array. Again, the c_7 term should should contribute 20% of the total parallel execution time for an efficiency of 0.8. The constribution of the c_7 term starts out at 22.1% on a 32 x 32 processor array and improves to 20.2% on a 1024 x 1024 processor array.

From the above discussion for GJE, the FFT-isoefficiency accuracy for the predicted problem size might be expected to improve as the processor-array size increases, but as Table 7.6 shows, the accuracy degrades as the processor-array size is increased. A more detailed

analysis is needed in the FFT case to explain this behavior. The exponential nature of equation 7.9 causes any error resulting from the asymptotic term's approximation to be amplified. While the asymptotic c_2 and c_7 terms improve in accuracy as P is increased, they are multipled by a $\sqrt{P}$ term that is also increasing. The improvement in the c_2 and c_7 asymptotic terms must be decreasing at a slower rate than the $\sqrt{P}$ multiplier, so the overall error increases.

7.4. Effects of Varying Machine Parameters on the Accuracy of the Isoefficiency-Scalability Function

The effects of varying the communcation and computation speeds of the computer architecture are very algorithm dependent. Figure 7.5 and Figure 7.6 show how the percentage error in the isoefficiency-scalability predictions for GJE vary with communication speed and computation speed, respectively. For FFT, the behavior in the isoefficiency-scalability predictions with varying communication and computation speed are extremely different than GJE's behavior. Figure 7.7 and Figure 7.8 show FFT's behavior.

As the communication speed is increased, the accuracy of the isoefficiency-scalability degrades quickly and saturates at nearly 100% error as the communication speed increases. The accuracy of the GJE problem-size prediction depends on how well the c_1 term approximates $T_1(N)$ and how well the c_4 term approximates $T_0(P, N)$. Table 7.9 shows that the contribution of the c_i terms as the communication speed is increased. The c_4 term drops from 3.1% with no speedup to 0.0% for the 100-times communcation speedup.

As the computation (and memory) speed is increased, all terms of the sequential execution time ($T_1(N)$) and all computation terms of the parallel overhead (T_0) decrease

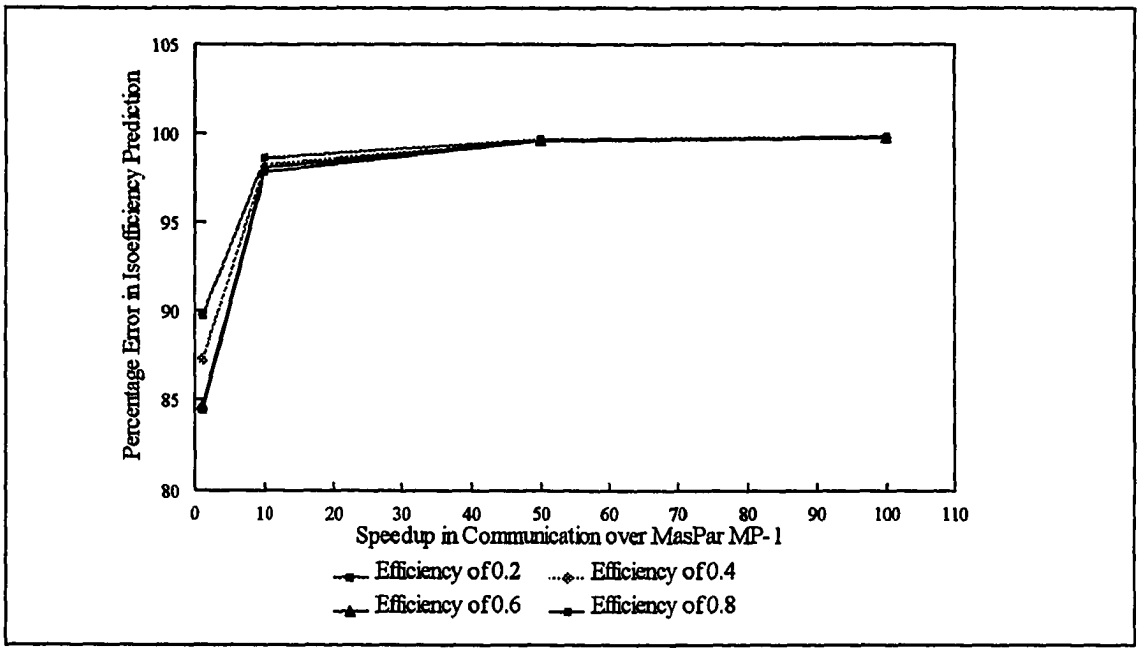

Figure 7.5. Degradation in the isoefficiency scalability accuracy for GJE as the communication speed is increased for a 128 x 128 processor array.

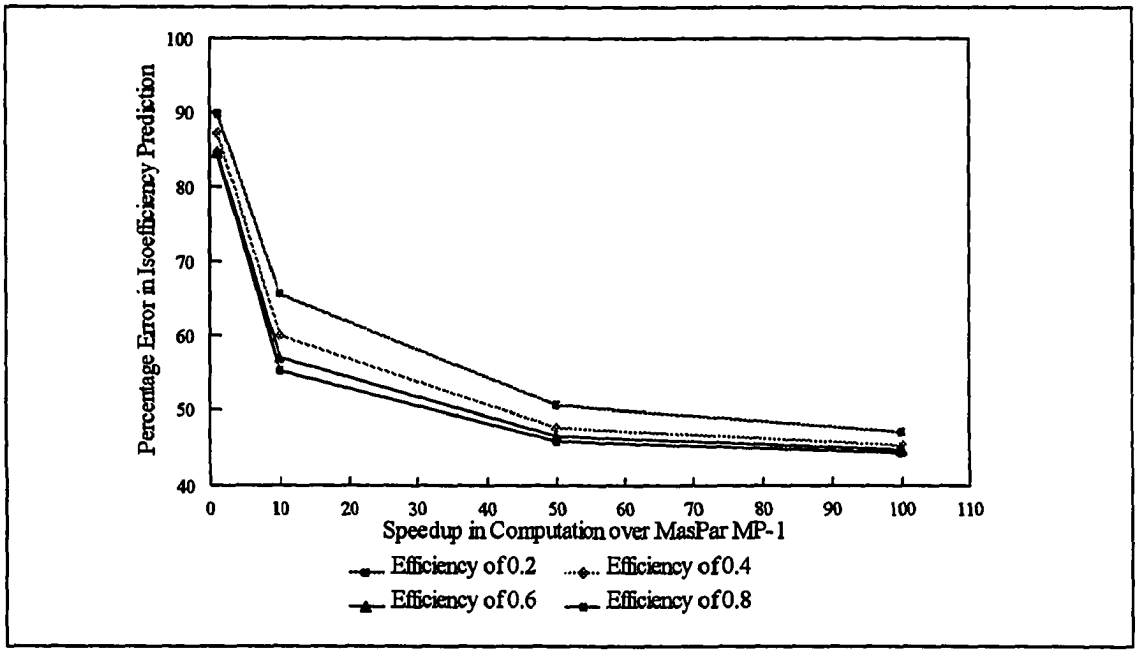

Figure 7.6. Improvement in the isoefficiency scalability accuracy for GJE as the compution speed is increased for a 128 x 128 processor array.

Table 7.9. Contribution of each term in the Gauss-Jordan Elimination to their respective sequential or parallel execution time for a constant efficiency of 0.8 on a 128 x 128 processor array as the communicaton speed is increased.

Comm. Speedup	Sequential Terms		Parallel Terms						
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
1	99.9%	0.1%	80.0%	3.1%	15.8%	0.1%	0.8%	0.2%	0.0%
10	99.9%	0.1%	79.9%	0.4%	18.6%	0.0%	0.7%	0.4%	0.0%
50	99.9%	0.1%	79.9%	0.1%	18.9%	0.0%	0.7%	0.4%	0.0%
100	99.9%	0.1%	79.9%	0.0%	19.0%	0.0%	0.7%	0.4%	0.0%

Table 7.10. Contribution of each term in the Gauss-Jordan Elimination to their respective sequential or parallel execution time for a constant efficiency of 0.8 on a 128 x 128 processor array as the computation speed is increased.

Comp. Speedup	Sequential Terms		Parallel Terms						
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
1	99.9%	0.1%	80.0%	3.1%	15.8%	0.1%	0.8%	0.2%	0.0%
10	100.0%	0.0%	80.0%	8.9%	10.5%	0.1%	0.4%	0.0%	0.0%
50	100.0%	0.0%	80.0%	10.8%	9.0%	0.0%	0.1%	0.0%	0.0%
100	100.0%	0.0%	80.0%	11.1%	8.8%	0.0%	0.1%	0.0%	0.0%

linearly with the increased speedup of the computation. So, the computation terms of the parallel execution time decrease relative to the asymptotic communication term (c_4 term). The combined effect causes a better than linear improvement in the isoefficiency-scalability predictions as the computational speedup is increased.

For FFT, only the isoefficiency function with efficiency = 0.8 was studied on a 128 x 128 processor array. At lower efficiencies, less than one element per processor was needed

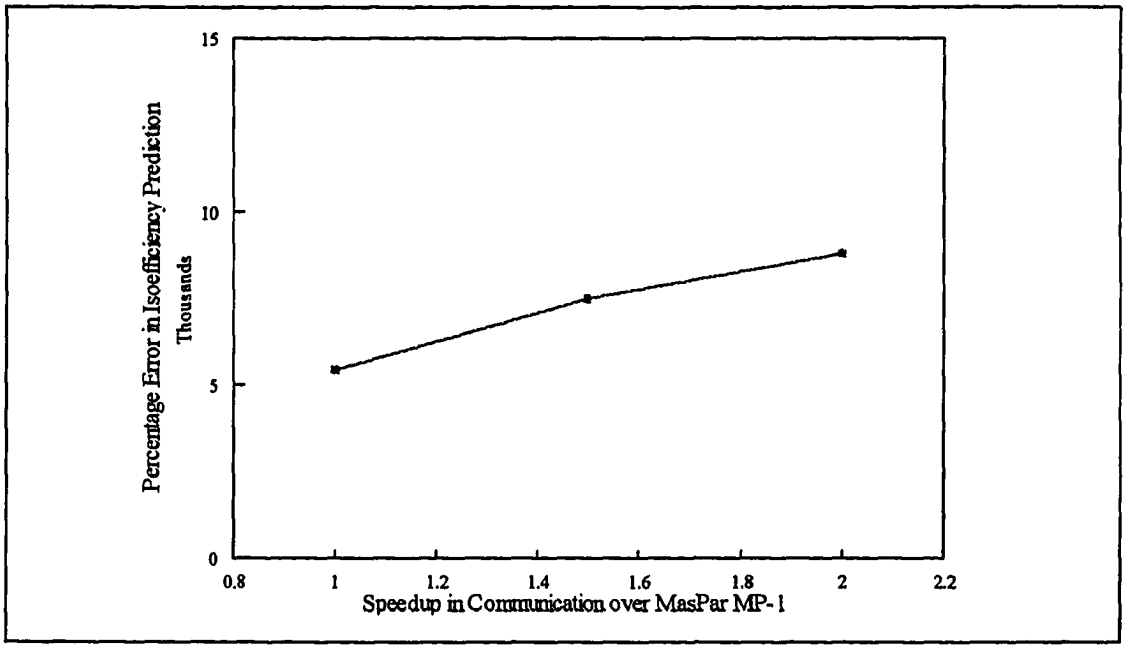

Figure 7.6. Degradation in the isoefficiency scalability accuracy (efficiency = 0.8) for FFT as the communication speed is increased for a 128x128 processor array.

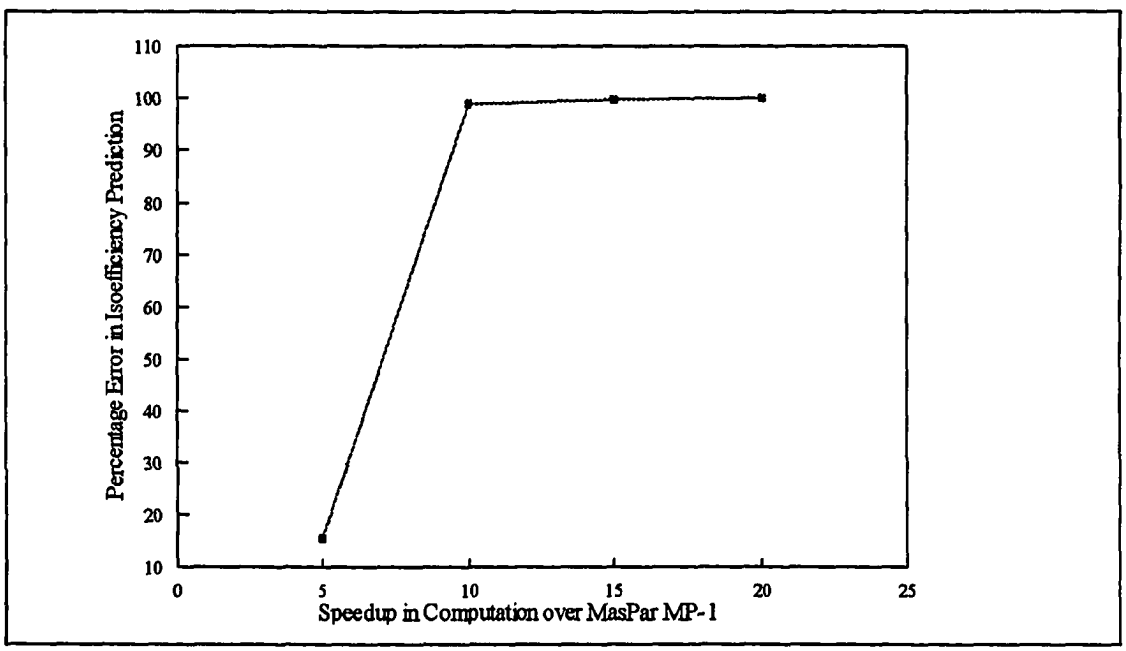

Figure 7.7. Degradation of isoefficiency-scalability accuracy (efficiency = 0.8) for FFT as the compution speed is increased for a 128 x 128 processor array.

Table 7.11. Contribution of each term in the Fast Fourier Transform to their respective sequential or parallel execution time for a constant efficiency of 0.8 as the communication speed is increased.

Communication Speedup	Sequential Terms		Parallel Terms				
	c_1	c_2	c_3	c_4	c_5	c_6	c_7
1.0	7.1%	92.9%	68.6%	3.8%	5.8%	0.6%	21.2%
1.5	11.1%	88.9%	65.6%	2.4%	9.1%	0.7%	22.2%
2.0	15.5%	84.5%	62.3%	0.9%	12.7%	0.7%	23.3%

to achieve an efficiency greater than 0.8. Similarly, only a one-and-one-half fold and two fold speedup of the communication speed was able to be studied. The results are summarized in Figure 7.6. When increasing the computation speed, the problem size quickly became larger than could be stored in a double precision number, so a maximum computation speedup of up to twenty fold was performed with results shown in Figure 7.7.

Table 7.11 shows the contribution of individual c_i terms as the communication speed is increased for a constant efficiency of 0.8. Both of the asymptotic terms, c_2 and c_7, diverge from the expected values of 100% and 20%, respectively, as the communication speed increases. Therefore, the isoefficiency-predicted problem size becomes less accurate as the communication speed increases.

As Table 7.12 shows, the asymptotic term c_2 approaches the expected values of 100% as the computation speed increases. However, the asymptotic c_7 term drops below 20% and remains at about 19.5% as the computation speed increases. This is primarily due to the other communication term, c_6 term, contributing 0.6% of the parallel-execution time. The c_6 term represents the startup time of the communications. The exponential nature of

Table 7.12. Contribution of each term in the Fast Fourier Transform to their respective sequential or parallel execution time for a constant efficiency of 0.8 as the computation speed is increased.

Computation Speedup	Sequential Terms		Parallel Terms				
	c_1	c_2	c_3	c_4	c_5	c_6	c_7
1	7.1%	92.9%	68.6%	3.8%	5.8%	0.6%	21.2%
5	1.3%	98.7%	72.8%	5.8%	1.1%	0.6%	19.7%
10	0.7%	99.3%	73.3%	6.0%	0.5%	0.6%	19.6%
15	0.4%	99.6%	73.5%	6.1%	0.4%	0.6%	19.5%
20	0.3%	99.7%	73.5%	6.1 %	0.3%	0.6%	19.5%

equation 7.9 amplifies the error introduced by the c_6 term as the computation speed is increased, because larger size problems are being solved.

CHAPTER 8. CONCLUSIONS

The asymptotic scalability metric, called Constant-Memory-per-Processor (CMP) scalability, was presented. This metric is useful in analyzing performance of a parallel algorithm on a distributed memory architecture as the number of processors grows, but the memory size per processor remains fixed. To illustrate the CMP scalability metric, parallel Matrix Multiplication (MM), Gauss Jordan Elimination (GJE), and Fast Fourier Transform (FFT) algorithms are considered on the hypercube and two-dimensional mesh topologies.

A comparison between the asymptotic CMP scalability and the isoefficiency scalability metrics is performed to gain a better understanding of scalability. An analysis of the scalability of GJE and FFT on a mesh predicts that GJE is asymptotically more scalable than FFT using the isoefficiency metric, but the CMP scalability metric predicts that FFT is asymptotically more scalable than GJE. Closer investigation reveals that both are correct, and that each metric corresponds to a different planer cross-section of a multidimensional performance surface.

By combining information from both the isoefficiency and CMP scalability metrics for two algorithms with conflicting isoefficiency and CMP scalability results, we are able to show how to predict the relative change in performance of the two algorithms along the fixed-processor planar cross-section and the fixed-problem size planar cross-section. Specifically, we showed that asymptotically (1) the algorithm that is more CMP scalable will experience a slower drop in efficiency as the number of processors is increased while keeping the problem size fixed, and (2) the algorithm that is more isoefficiency scalable will

experience a greater increase in efficiency as the problem size increases for a fixed number of processors.

Two classes of algorithms are shown to be not fixed-time scalable, i.e., there is a maximum size problem for any fixed time such that larger size problems can not be completed within that time. These classes of algorithms include (1) algorithms containing a section of code where the number of times it executes is a monotonically increasing function of P, and (2) algorithms where information flows between two processors whose distance is a monotonically increasing function of P. These classes of algorithms covering the majority of conceivable algorithms.

Scalability metrics such as the CMP-scalability metric and isoefficiency metric indicate the asymptotic behavior as the number of processors becomes large. However, we question how useful these metrics are on a specific machine with a fixed number of processors and memory per processor. Our investigation of the utility of the CMP and the isoefficiency-scalability metrics for the three algorithms on a 16K processor MasPar MP-1 machine showed that: (1) the CMP scalability metric was a poor predictor of performance especially for the computationally intensive algorithms, (2) a 10-fold increase in the computational speed greatly improved the CMP scalability accuracy, (3) the isoefficiency metric was a poor predictor of the problem size necessary to achieve a specified efficiency, and (4) improvements in computation speed improved the accuracy of isoefficiency for GJE, but degraded its accuracy for FFT.

In general caution would be recommended when trying to apply either of these

metrics. The machine specific constants must be examined for an algorithm if these scalability metrics are to be used. The critical factors in determining the applicability of these metrics is the ratio of the sequential execution time to the asymptotically identified computation term of the sequential execution time, and the ratio of the parallel execution time to the asymptotically identified communication term of the parallel execution time. The closer these ratios are to one, the better the scalability metric will apply.

8.2. Further Work

Theorems 4.1 and 4.2 predict the relative change in performance of the two algorithms along the fixed-processor planar cross-section and the fixed-problem size planar cross-section by combining information from both the isoefficiency and CMP scalability metrics if the two algorithms have conflicting isoefficiency and CMP scalability results. Quantification of these results might be possible with further work.

Not all algorithms have a one dimensional size component, N, to predict the problem size, it would be interesting to study such an algorithm to see how the ideas of scalability metrics apply.

REFERENCES

[1] Alpern, B. and Carter L., "Is Scalability Relevant: A Look at Sparse Matrix-Vector Product," *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, (1995), pp. 850-851.

[2] Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS Conference Proceeding*, (April, 1967), pp. 483-485.

[3] Bailey, D. H., "Misleading Performance in the Supercomputing Field," *Proceedings in Supercomputing '92*, (1992), pp. 155-158.

[4] Cannon, L. E., "A Cellular Computer to Implement the Kalman Filter," Ph.D. Thesis, Montana State University, Bozeman, MT, (1969).

[5] Fienup, M., and Kothari, S.C. , "Implementations of Fast Fourier Transform on the MasPar MP-1 and MP-2," Technical Report TR 93-01, Computer Science Department, Iowa State University, Ames, IA 50011, (1993).

[6] Fienup, M. and Kothari, S.C., "CMP: A Memory-Constrained Scalability Metric," *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, (1995), pp. 848-849.

[7] Fienup, M. and Kothari, S.C., "A Memory Constrained Scalability Metric," *Proceedings of the 1994 International Conference on Parallel Processing*, vol. III, (1994), pp. 1-7.

[8] Flatt, H. P. and Kennedy, K., "Performance of Parallel Processors," *Parallel Computing*, vol. 12, (1989), pp. 1-20.

[9] Grama, A., Grupta, A., and Kumar, V., "Isoefficiency function: A scalability metric for parallel algorithms and architectures," Technical Report TR 93-24, Computer Science Department, University of Minnesota, Minneapolis, MN, (1993).

[10] Gupta, A., and Kumar, V., "The Scalability of FFT on Parallel Computers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, (August, 1993), pp. 922-932.

[11] Gupta, A., and Kumar , V., "Scalability of Parallel Algorithms for Matrix Multiplication," *Proceedings of the 1993 International Conference on Parallel Processing*, vol. III, (1993), pp. 115-123.

[12] Gupta, A., and Kumar , V., "Performance Properties of Large Scale Parallel

Systems," *Journal of Parallel and Distributed Computing*, vol. 19, (November, 1993).

[13] Gustafson, J. L., Montry, G. R., and Benner, R. E., "Development of parallel methods for a 1024-processor hypercube," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, (April, 1988), pp. 609-638.

[14] Gustafson , J. L., "Reevaluating Amdahl's Law," *Commun. ACM*, vol. 31, (August, 1988), pp. 532-533.

[15] Gustafson, J. L., "The consequences of fixed time performance measurement," *Proceedings of the 25th Hawaii International Conference on System Sciences: Volume III*, (1992), pp. 113-124.

[16] Kumar, V., and Gupta, A., "Analyzing Scalability of Parallel Algorithms and Architectures," Tech. Rep. TR-91-18, Computer Science Department., University of Minnesota, Minneapolis, MN, (June, 1991).

[17] Kumar, V., Grama, A. Y., Gupta, A., and Karypis, G., *Introduction to Parallel Computing, Design and Analysis of Algorithms*, Benjamin/Cummings, Redwood City, CA, (1994).

[18] Kumar, V., and Gupta, A., "Analyzing Scalability of Parallel Algorithms and Architectures," Tech. Rep. TR-91-18, Comput. Sci. Dept., Univ. of Minnesota, Minneapolis, MN, (June, 1991).

[19] Kumar, V., and Rao, V. N., "Parallel depth-first search, part II: Analysis," *International Journal of Parallel Programming*, vol. 16, (June, 1987), pp. 501-519.

[20] Kung, H. T., "Memory requirements for balanced computer architectures," *Proceedings of the 1986 IEEE Symposium on Computer Architecture*, (1986), pp. 49-54.

[21] Lam, M. S., "Software Pipelining: An Effective Scheduling Techique for VLIW Machines," *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, (1988), pp. 318-328.

[22] Marinescu, D. C. and Rice, J. R., "On high level characterization of parallelism," Technical Report CSD-TR-1011, CAPO Report CER-90-32, Computer Science Department, Prudue University, West Lafayette, IN, (June, 1991).

[23] MasPar Assembly Language Reference Manual, MasPar Computer Corportation, Sunnyvale, CA, (1990).

[24] Singh, V., Hennessy, J. L., and Gupta, A., "Scaling parallel programs for multiprocessos: Methodology and examples," *IEEE Computer*, vol. 20 (July, 1993), pp. 42-50.

[25] Sun, X. H., and Gustafson, J. L., "Toward a better parallel performance metric," *Parallel Computing*, vol. 17, (1991), pp. 1093-1109.

[26] Sun, X. H., and Ni L. M., "Scalable Problems and Memory-bounded Speedup," *Journal of Parallel and Distributed Computing*, vol. 19, (September, 1993), pp. 27-37.

[26] Sun, X. H., and Rover, D. T., "Scalability of Parallel Algorithm-Machine Combinations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, (June, 1994), pp. 599-613.

[27] Singh, J. P., Hennessy, J. L., and Goopta, A. "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *Computer*, vol. 26, (July, 1993), pp. 42-50.

[28] Tong, C., and Swarztrauber, P. N., "Ordered Fast Fourier Transforms on a Massively Parallel Hypercube Multiprocessor," *Journal of Parallel and Distributed Computing*, vol. 12, (Janurary, 1991), pp. 50-59.

[29] Worley, P. H., "The effect of time constraints on scaled speedup," *SIAM Journal on Scientific and Statistical Computing*, vol. 11, (May, 1990), pp. 838-858.

ACKNOWLEDGEMENTS

I would like to thank Ames Laboratory for allowing me access to their MasPar MP-1 and MP-2 computers. Additionally, I would like to thank the professors on my committee.

APPENDIX A. ADDITIONAL TABLES

Table A.1. Matrix multiplication timings on a 32 x 32 MasPar MP-1.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
32 x 32	64 x 64	4	0.0152	544.7	0.5319
32 x 32	128 x 128	16	0.0931	711.2	0.6945
32 x 32	256 x 256	64	0.6286	842.5	0.8227
32 x 32	384 x 384	144	1.9980	894.5	0.8736
32 x 32	512 x 512	256	4.5850	923.9	0.9023
32 x 32	640 x 640	400	8.7773	942.6	0.9206
32 x 32	768 x 768	576	14.9440	956.7	0.9343
32 x 32	896 x 896	784	23.5190	965.3	0.9427
32 x 32	1024 x 1024	1024	34.8513	972.4	0.9496

Table A.2. Matrix multiplication timings on a 64 x 64 MasPar MP-1.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
64 x 64	128 x 128	4	0.0304	2178.0	0.5317
64 x 64	256 x 256	16	0.1863	2842.7	0.6940
64 x 64	512 x 512	64	1.2571	3369.9	0.8227
64 x 64	768 x 768	144	3.9960	3577.8	0.8735
64 x 64	1024 x 1024	256	9.1701	3695.6	0.9022
64 x 64	1280 x 1280	400	17.5547	3770.4	0.9205
64 x 64	1536 x 1536	576	29.8880	3826.7	0.9343
64 x 64	1792 x 1792	784	47.0380	3861.1	0.9427
64 x 64	2048 x 2048	1024	69.7025	3889.5	0.9496

Table A.3. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer with 100 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	477.3%	517.6%	582.3%	679.5%	820.8%	1023.5%
64	244.4%	262.8%	294.0%	341.8%	411.9%	512.8%
128	125.9%	134.1%	148.9%	172.2%	206.9%	257.1%
256	65.8%	69.0%	75.9%	87.1%	104.2%	129.0%
512	35.3%	36.2%	39.1%	44.4%	52.7%	64.9%
1,024	19.8%	19.6%	20.6%	23.0%	26.9%	32.9%
2,048	11.9%	11.2%	11.3%	12.2%	13.9%	16.8%
4,096	7.9%	7.0%	6.7%	6.8%	7.5%	8.7%
8,192	5.9%	4.9%	4.3%	4.1%	4.2%	4.7%
16,384	4.9%	3.8%	3.1%	2.7%	2.6%	2.7%

Table A.4. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer with 10 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	48.6%	52.6%	56.5%	60.5%	64.3%	68.2%
64	21.6%	23.8%	26.0%	28.1%	30.2%	32.3%
128	6.9%	8.2%	9.4%	10.7%	11.9%	13.0%
256	-0.9%	-0.1%	0.7%	1.4%	2.2%	2.9%
512	-4.9%	-4.3%	-3.8%	-3.3%	-2.8%	-2.3%
1,024	-6.8%	-6.4%	-6.0%	-5.6%	-5.3%	-4.9%
2,048	-7.6%	-7.3%	-7.0%	-6.7%	-6.4%	-6.1%
4,096	-7.8%	-7.6%	-7.3%	-7.1%	-6.8%	-6.6%
8,192	-7.8%	-7.5%	-7.3%	-7.1%	-6.9%	-6.7%
16,384	-7.6%	-7.4%	-7.2%	-7.0%	-6.8%	-6.7%

Table A.5. Gauss-Jordan Elimination timings on a 64 x 64 MasPar MP-1.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
64 x 64	128 x 128	4	0.1045	331.1	0.0808
64 x 64	192 x 192	9	0.1980	585.9	0.1430
64 x 64	256 x 256	16	0.3273	837.5	0.2045
64 x 64	512 x 512	64	1.3303	1640.4	0.4005
64 x 64	768 x 768	144	3.4134	2154.4	0.5260
64 x 64	1024 x 1024	256	6.9866	2492.9	0.6086
64 x 64	1280 x 1280	400	12.4245	2736.6	0.6681
64 x 64	1534 x 1534	576	20.1598	2913.5	0.7113
64 x 64	1792 x 1792	784	30.5799	3049.3	0.7445
64 x 64	2048 x 2048	1024	44.0875	3156.6	0.7707
64 x 64	2304 x 2304	1296	61.0900	3243.2	0.7918
64 x 64	2560 x 2560	1600	81.9815	3314.7	0.8093
64 x 64	2816 x 2816	1936	107.1683	3374.7	0.8239
64 x 64	3072 x 3072	2304	137.0559	3425.6	0.8363
64 x 64	3328 x 3328	2704	172.0354	3469.6	0.8471
64 x 64	3584 x 3584	3136	212.5180	3507.8	0.8564

Table A.6. Gauss-Jordan Elimination timings on a 128 x 128 MasPar MP-1.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
128 x 128	256 x 256	4	0.2226	1231.1	0.0751
128 x 128	384 x 384	9	0.4204	2193.7	0.1339
128 x 128	512 x 512	16	0.6919	3154.1	0.1925
128 x 128	1024 x 1024	64	2.7727	6281.5	0.3834
128 x 128	1536 x 1536	144	7.0504	8330.7	0.5085
128 x 128	2048 x 2048	256	14.3368	9707.0	0.5925
128 x 128	2560 x 2560	400	25.4089	10695.0	0.6528
128 x 128	3072 x 3072	576	41.0963	11424.5	0.6973
128 x 128	3584 x 3584	784	62.1994	11985.2	0.7315
128 x 128	4096 x 4096	1024	89.5139	12430.2	0.7587
128 x 128	4608 x 4608	1296	123.8493	12791.0	0.7807
128 x 128	5120 x 5120	1600	166.0106	13089.1	0.7989
128 x 128	5632 x 5632	1936	216.7853	13340.6	0.8142
128 x 128	6144 x 6144	2304	277.0066	13553.9	0.8273
128 x 128	6656 x 6656	2704	347.4602	13738.0	0.8385
128 x 128	7168 x 7168	3136	428.9462	13898.5	0.8483

Table A.7. Fast Fourier Transform timings on a 32 x 32 MasPar MP-1.

Processor Array	Log_s(Number of Elements)	Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
32 x 32	12	4	0.0103	734.3	0.7171
32 x 32	13	8	0.0216	748.5	0.7309
32 x 32	14	16	0.0451	759.3	0.7415
32 x 32	15	32	0.0944	768.2	0.7502
32 x 32	16	64	0.1975	775.3	0.7571
32 x 32	17	128	0.4123	781.4	0.7631
32 x 32	18	256	0.8597	786.8	0.7684
32 x 32	19	512	1.7899	791.6	0.7731
32 x 32	20	1024	3.7210	796.0	0.7774

Table A.8. Fast Fourier Transform timings on a 64 x 64 MasPar MP-1.

Processor Array	Log_s(Number of Elements)	Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
64 x 64	14	4	0.0128	2662.3	0.6500
64 x 64	15	8	0.0267	2719.5	0.6639
64 x 64	16	16	0.0554	2764.5	0.6749
64 x 64	17	32	0.1150	2802.9	0.6843
64 x 64	18	64	0.2385	2836.3	0.6925
64 x 64	19	128	0.4944	2866.1	0.6997
64 x 64	20	256	1.0238	2893.1	0.7063
64 x 64	21	512	2.1180	2917.9	0.7124
64 x 64	22	1024	4.3773	2940.8	0.7180

Table A.9. Fast Fourier Transform timings on a 128 x 128 MasPar MP-1.

Processor Array	Log_s(Number of Elements)	Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
128 x 128	16	4	0.0168	9102.2	0.5556
128 x 128	17	8	0.0346	9320.0	0.5688
128 x 128	18	16	0.0712	9505.6	0.5802
128 x 128	19	32	0.1465	9671.8	0.5903
128 x 128	20	64	0.3016	9821.8	0.5995
128 x 128	21	128	0.6205	9959.3	0.6079
128 x 128	22	256	1.2761	10087.3	0.6157
128 x 128	23	512	2.6227	10207.1	0.6230
128 x 128	24	1024	5.3866	10319.9	0.6299

Table A.10. Matrix Multiplication timings on a 32 x 32 MasPar MP-2.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
32 x 32	64 x 64	4	0.0051	354.5	0.3462
32 x 32	128 x 128	16	0.0275	525.5	0.5132
32 x 32	256 x 256	64	0.1698	680.6	0.6647
32 x 32	384 x 384	144	0.5201	749.8	0.7323
32 x 32	512 x 512	256	1.1661	792.7	0.7741
32 x 32	640 x 640	400	2.2050	818.7	0.7996
32 x 32	768 x 768	576	3.7159	839.5	0.8198
32 x 32	896 x 896	784	5.8017	853.8	0.8338
32 x 32	1024 x 1024	1024	8.5543	864.4	0.8441
32 x 32	1152 x 1152	1296	12.0623	872.8	0.8523
32 x 32	1280 x 1280	1600	16.4306	878.9	0.8583
32 x 32	1408 x 1408	1936	21.7065	885.5	0.8648
32 x 32	1536 x 1536	2304	28.0051	891.1	0.8702
32 x 32	1664 x 1664	2704	35.4172	895.8	0.8748
32 x 32	1792 x 1792	3136	44.0914	898.7	0.8777
32 x 32	1920 x 1920	3600	54.0267	902.1	0.8810
32 x 32	2048 x 2048	4096	65.4024	904.4	0.8832

Table A.11. Matrix Multiplication timings on a 64 x 64 MasPar MP-2.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
64 x 64	128 x 128	4	0.0102	1416.8	0.3459
64 x 64	256 x 256	16	0.0551	2097.4	0.5121
64 x 64	512 x 512	64	0.3395	2722.7	0.6647
64 x 64	768 x 768	144	1.0403	2998.7	0.7321
64 x 64	1024 x 1024	256	2.3321	3170.6	0.7741
64 x 64	1280 x 1280	400	4.4099	3274.8	0.7995
64 x 64	1536 x 1536	576	7.4318	3357.8	0.8198
64 x 64	1792 x 1792	784	11.6034	3415.1	0.8338
64 x 64	2048 x 2048	1024	17.1086	3457.4	0.8441
64 x 64	2304 x 2304	1296	24.1246	3491.1	0.8523
64 x 64	2560 x 2560	1600	32.8612	3515.6	0.8583
64 x 64	2816 x 2816	1936	43.4130	3542.0	0.8647
64 x 64	3072 x 3072	2304	56.0103	3564.2	0.8702
64 x 64	3328 x 3328	2704	70.8343	3583.2	0.8748
64 x 64	3584 x 3584	3136	88.1828	3594.9	0.8777
64 x 64	3840 x 3840	3600	108.0534	3608.4	0.8810
64 x 64	4096 x 4096	4096	130.8049	3617.6	0.8832

Table A.12. Gauss-Jordan Elimination timings on a 32 x 32 MasPar MP-2.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
32 x 32	64 x 64	4	0.0229	48.3	0.0472
32 x 32	96 x 96	9	0.0410	89.6	0.0875
32 x 32	128 x 128	16	0.0648	133.2	0.1301
32 x 32	256 x 256	64	0.2328	293.0	0.2861
32 x 32	384 x 384	144	0.5574	411.3	0.4017
32 x 32	512 x 512	256	1.1016	492.3	0.4808
32 x 32	640 x 640	400	1.8891	560.0	0.5468
32 x 32	768 x 768	576	3.0024	608.3	0.5941
32 x 32	896 x 896	784	4.4842	646.4	0.6312
32 x 32	1024 x 1024	1024	6.3887	676.9	0.6611
32 x 32	1152 x 1152	1296	8.7684	702.0	0.6856
32 x 32	1280 x 1280	1600	11.6805	722.7	0.7058
32 x 32	1408 x 1408	1936	15.1791	740.0	0.7227
32 x 32	1536 x 1536	2304	19.3157	754.9	0.7372
32 x 32	1664 x 1664	2704	24.1557	767.3	0.7493
32 x 32	1792 x 1792	3136	29.7353	778.4	0.7602

Table A.13. Gauss-Jordan Elimination timings on a 64 x 64 MasPar MP-2.

Processor Array	Array Size (N x N)	Array Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
64 x 64	128 x 128	4	0.0271	174.2	0.0425
64 x 64	192 x 192	9	0.0447	326.0	0.0796
64 x 64	256 x 256	16	0.0645	488.3	0.1192
64 x 64	512 x 512	64	0.1743	1097.1	0.2678
64 x 64	768 x 768	144	0.3383	1558.6	0.3805
64 x 64	1024 x 1024	256	0.5644	1888.0	0.4609
64 x 64	1280 x 1280	400	0.8624	2152.1	0.5254
64 x 64	1534 x 1534	576	1.2379	2346.3	0.5728
64 x 64	1792 x 1792	784	1.7028	2500.7	0.6105
64 x 64	2048 x 2048	1024	2.2627	2625.4	0.6410
64 x 64	2304 x 2304	1296	2.9262	2728.7	0.6662
64 x 64	2560 x 2560	1600	3.7036	2815.9	0.6875
64 x 64	2816 x 2816	1936	4.6004	2889.5	0.7054
64 x 64	3072 x 3072	2304	5.6279	2952.0	0.7207
64 x 64	3328 x 3328	2704	6.7931	3006.8	0.7341
64 x 64	3584 x 3584	3136	8.1043	3053.9	0.7456
64 x 64	3840 x 3840	3600	9.5684	3094.7	0.7555
64 x 64	4096 x 4096	4096	10.7004	3131.5	0.7645
64 x 64	4352 x 4352	4624	12.4360	3164.4	0.7726
64 x 64	4608 x 4608	5184	14.3476	3193.7	0.7797
64 x 64	4864 x 4864	5776	16.4430	3219.9	0.7861
64 x 64	5120 x 5120	6400	18.7307	3243.7	0.7919
64 x 64	5376 x 5376	7056	21.2197	3265.6	0.7973
64 x 64	5632 x 5632	7744	23.9158	3285.3	0.8021
64 x 64	5888 x 5888	8464	26.8306	3303.5	0.8065
64 x 64	6144 x 6144	9216	29.9716	3320.2	0.8106
64 x 64	6400 x 6400	10000	33.6470	3335.7	0.8144
64 x 64	6656 x 6656	10816	36.9637	3350.0	0.8179
64 x 64	6912 x 6912	11664	40.8327	3363.2	0.8211
64 x 64	7168 x 7168	12544	44.95872	3375.4	0.8241

Table A.14. Fast Fourier Transform timings on a 32 x 32 MasPar MP-2.

Processor Array	Log_s(Number of Elements)	Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
32 x 32	12	4	0.0035	494.5	0.4829
32 x 32	13	8	0.0072	511.2	0.4992
32 x 32	14	16	0.0147	525.7	0.5133
32 x 32	15	32	0.0304	537.5	0.5249
32 x 32	16	64	0.0627	548.2	0.5353
32 x 32	17	128	0.1292	557.8	0.5447
32 x 32	18	256	0.2663	566.6	0.5533
32 x 32	19	512	0.5484	574.7	0.5612
32 x 32	20	1024	1.1287	582.3	0.5686
32 x 32	21	2048	2.3213	589.4	0.5756
32 x 32	22	4096	4.7707	596.1	0.5822

Table A.15. Fast Fourier Transform timings on a 64 x 64 MasPar MP-2.

Processor Array	Log_s(Number of Elements)	Elements per Processor	Parallel Execution Time (seconds)	Speedup	Efficiency
64 x 64	14	4	0.0051	1518.0	0.3706
64 x 64	15	8	0.0104	1573.0	0.3840
64 x 64	16	16	0.0212	1620.8	0.3957
64 x 64	17	32	0.0433	1664.9	0.4065
64 x 64	18	64	0.0885	1705.3	0.4163
64 x 64	19	128	0.1808	1743.3	0.4256
64 x 64	20	256	0.3694	1779.0	0.4343
64 x 64	21	512	0.7547	1812.9	0.4426
64 x 64	22	1024	1.5413	1845.2	0.4505
64 x 64	23	2048	3.1466	1876.1	0.4580
64 x 64	24	4096	6.4212	1905.7	0.4652

Table A.16. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer with 10 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	1751.6%	2253.0%	2970.8%	3992.1%	5440.8%	7492.5%
64	882.5%	1131.5%	1489.1%	1998.9%	2722.6%	3748.1%
128	445.3%	568.6%	746.7%	1001.0%	1362.5%	1874.9%
256	225.6%	286.4%	374.8%	501.6%	682.0%	938.0%
512	115.2%	144.9%	188.6%	251.6%	341.6%	469.4%
1024	59.8%	74.0%	95.4%	126.6%	171.3%	235.1%
2048	31.9%	38.4%	48.7%	64.0%	86.2%	117.9%
4096	17.9%	20.6%	25.3%	32.7%	43.6%	59.3%
8192	10.9%	11.7%	13.7%	17.0%	22.3%	30.0%
16,384	7.4%	7.2%	7.8%	9.2%	11.6%	15.3%

Table A.17. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer with 50 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	618.9%	710.4%	847.7%	1047.6%	1334.2%	1742.3%
64	315.3%	359.3%	426.8%	525.9%	668.6%	872.3%
128	161.4%	182.4%	215.3%	264.3%	335.3%	436.8%
256	83.5%	93.2%	109.1%	133.2%	168.4%	218.9%
512	44.1%	48.3%	55.7%	67.4%	84.8%	109.9%
1024	24.2%	25.7%	28.9%	34.5%	42.9%	55.3%
2048	14.1%	14.3%	15.5%	18.0%	22.0%	28.0%
4096	9.0%	8.5%	8.7%	9.7%	11.5%	14.4%
8192	6.5%	5.6%	5.3%	5.5%	6.2%	7.5%
16,384	5.2%	4.2%	3.6%	3.4%	3.6%	4.1%

Table A.18. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer with 100 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	477.3%	517.6%	582.3%	679.5%	820.8%	1023.5%
64	244.4%	262.8%	294.0%	341.8%	411.9%	512.8%
128	125.9%	134.1%	148.9%	172.2%	206.9%	257.1%
256	65.8%	69.0%	75.9%	87.1%	104.2%	129.0%
512	35.3%	36.2%	39.1%	44.4%	52.7%	64.9%
1024	19.8%	19.6%	20.6%	23.0%	26.9%	32.9%
2048	11.9%	11.2%	11.3%	12.2%	13.9%	16.8%
4096	7.9%	7.0%	6.7%	6.8%	7.5%	8.7%
8192	5.9%	4.9%	4.3%	4.1%	4.2%	4.7%
16,384	4.9%	3.8%	3.1%	2.7%	2.6%	2.7%

Table A.19. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer with 10 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	48.6%	52.6%	56.5%	60.5%	64.3%	68.2%
64	21.6%	23.8%	26.0%	28.1%	30.2%	32.3%
128	6.9%	8.2%	9.4%	10.7%	11.9%	13.0%
256	-0.9%	-0.1%	0.7%	1.4%	2.2%	2.9%
512	-4.9%	-4.3%	-3.8%	-3.3%	-2.8%	-2.3%
1024	-6.8%	-6.4%	-6.0%	-5.6%	-5.3%	-4.9%
2048	-7.6%	-7.3%	-7.0%	-6.7%	-6.4%	-6.1%
4096	-7.8%	-7.6%	-7.3%	-7.1%	-6.8%	-6.6%
8192	-7.8%	-7.5%	-7.3%	-7.1%	-6.9%	-6.7%
16,384	-7.6%	-7.4%	-7.2%	-7.0%	-6.8%	-6.7%

Table A.20. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer with 50 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	-1.6%	-0.3%	1.0%	2.1%	3.3%	4.4%
64	-6.0%	-5.2%	-4.4%	-3.6%	-2.8%	-2.1%
128	-8.2%	-7.6%	-7.0%	-6.4%	-5.9%	-5.4%
256	-9.1%	-8.6%	-8.2%	-7.7%	-7.3%	-7.0%
512	-9.3%	-8.9%	-8.5%	-8.2%	-7.9%	-7.6%
1024	-9.1%	-8.8%	-8.5%	-8.2%	-8.0%	-7.7%
2048	-8.8%	-8.6%	-8.3%	-8.1%	-7.8%	-7.6%
4096	-8.5%	-8.2%	-8.0%	-7.8%	-7.6%	-7.4%
8192	-8.1%	-7.9%	-7.7%	-7.5%	-7.3%	-7.1%
16,384	-7.7%	-7.5%	-7.4%	-7.2%	-7.0%	-6.9%

Table A.21. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer with 100 times faster processors.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	-7.8%	-6.9%	-6.0%	-5.1%	-4.3%	-3.5%
64	-9.5%	-8.8%	-8.2%	-7.6%	-7.0%	-6.4%
128	-10.1%	-9.5%	-9.1%	-8.6%	-8.1%	-7.7%
256	-10.1%	-9.7%	-9.3%	-8.9%	-8.5%	-8.2%
512	-9.8%	-9.5%	-9.1%	-8.8%	-8.5%	-8.2%
1024	-9.4%	-9.1%	-8.8%	-8.6%	-8.3%	-8.1%
2048	-9.0%	-8.7%	-8.5%	-8.2%	-8.0%	-7.8%
4096	-8.6%	-8.3%	-8.1%	-7.9%	-7.7%	-7.5%
8192	-8.1%	-7.9%	-7.7%	-7.5%	-7.4%	-7.2%
16,384	-7.8%	-7.6%	-7.4%	-7.2%	-7.1%	-6.9%

Table A.22. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer with 10 times faster communication.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	141920.9%	193155.2%	265708.2%	368382.2%	513633.6%	719084.5%
64	71074.7%	96680.1%	132948.3%	184279.5%	256901.0%	359623.6%
128	35574.6%	48371.8%	66502.0%	92164.9%	128473.7%	179833.6%
256	17801.6%	24197.4%	33260.5%	46090.5%	64243.9%	89923.2%
512	8907.6%	12103.9%	16634.2%	23048.4%	32124.5%	44963.7%
1024	4457.9%	6054.9%	8319.3%	11525.8%	16063.5%	22482.8%
2048	2231.8%	3029.5%	4161.1%	5764.0%	8032.5%	11242.0%
4096	1118.3%	1516.5%	2081.8%	2882.9%	4016.9%	5621.4%
8192	561.3%	759.8%	1042.0%	1442.2%	2009.0%	2811.1%
16,384	282.7%	381.3%	522.0%	721.8%	1005.0%	1405.9%

Table A.23. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer with 50 times faster communication.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	708261%	964477%	1327273%	1840665%	2566938%	3594203%
64	354679%	482735%	664096%	920767%	1283885%	1797504%
128	177511%	241516%	332180%	460504%	642055%	898859%
256	88816%	120807%	166132%	230288%	321060%	449459%
512	44433%	60423%	83081%	115157%	160540%	224739%
1024	22228%	30220%	41547%	57583%	80274%	112373%
2048	11120%	15115%	20777%	28794%	40139%	56188%
4096	5564%	7560%	10391%	14399%	20071%	28095%
8192	2785%	3782%	5197%	7200%	10036%	14048%
16,384	1395%	1893%	2600%	3601%	5019%	7024%

Table A.24. Percentage error of CMP scalability function for Gauss-Jordan Elimination on a MasPar MP-1 computer with 100 times faster communication.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	1416187%	1928629%	2654230%	3681019%	5133568%	7188102%
64	709185%	965304%	1328032%	1841375%	2567614%	3594855%
128	354932%	482946%	664278%	920927%	1284031%	1797640%
256	177584%	241570%	332221%	460535%	642080%	898879%
512	88839%	120822%	166140%	230292%	321060%	449457%
1024	44441%	60427%	83082%	115155%	160538%	224735%
2048	22231%	30221%	41547%	57582%	80272%	112370%
4096	11122%	15115%	20777%	28793%	40138%	56186%
8192	5565%	7560%	10390%	14398%	20070%	28094%
16,384	2785%	3782%	5197%	7200%	10036%	14047%

Table A.25. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer with 10 times faster communication.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	6256.6%	6596.6%	6936.7%	7276.8%	7617.0%	7957.2%
64	3438.7%	3608.9%	3779.1%	3949.3%	4119.6%	4289.8%
128	1871.9%	1957.2%	2042.4%	2127.7%	2212.9%	2298.2%
256	1009.8%	1052.6%	1095.3%	1138.1%	1180.9%	1223.6%
512	539.5%	561.1%	582.6%	604.1%	625.6%	647.1%
1024	284.9%	295.8%	306.7%	317.6%	328.4%	339.3%
2048	148.0%	153.6%	159.1%	164.7%	170.2%	175.7%
4096	74.9%	77.8%	80.6%	83.5%	86.3%	89.2%
8192	36.0%	37.6%	39.1%	40.6%	42.1%	43.6%
16,384	15.6%	16.4%	17.3%	18.1%	18.9%	19.7%

Table A.26. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer with 50 times faster communication.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	31339.5%	33037.0%	34735.1%	36433.6%	38132.6%	39832.0%
64	17245.5%	18094.3%	18943.4%	19792.7%	20642.3%	21492.0%
128	9407.3%	9831.9%	10256.6%	10681.4%	11106.2%	11531.2%
256	5093.2%	5305.7%	5518.2%	5730.7%	5943.3%	6155.9%
512	2739.0%	2845.4%	2951.8%	3058.2%	3164.6%	3271.0%
1024	1463.5%	1516.9%	1570.2%	1623.5%	1676.8%	1730.1%
2048	776.7%	803.5%	830.3%	857.0%	883.8%	910.5%
4096	408.9%	422.4%	435.9%	449.4%	462.8%	476.3%
8192	212.9%	219.7%	226.6%	233.4%	240.2%	247.0%
16,384	108.9%	112.4%	115.9%	119.4%	122.9%	126.4%

Table A.26. Percentage error of CMP scalability function for Fast Fourier Transform on a MasPar MP-1 computer with 100 times faster communication.

$\sqrt{P}$	Memory Used Per Processor, β					
	1 K	2 K	4 K	8 K	16 K	32 K
32	62693.1%	66087.5%	69483.1%	72879.7%	76277.2%	79675.5%
64	34503.9%	36201.1%	37898.8%	39597.0%	41295.6%	42994.7%
128	18826.5%	19675.3%	20524.2%	21373.4%	22222.8%	23072.4%
256	10197.6%	10622.1%	11046.7%	11471.4%	11896.3%	12321.1%
512	5488.4%	5700.8%	5913.3%	6125.8%	6338.3%	6550.9%
1024	2936.8%	3043.1%	3149.5%	3255.9%	3362.2%	3468.6%
2048	1562.6%	1615.9%	1669.2%	1722.5%	1775.8%	1829.1%
4096	826.5%	853.2%	880.0%	906.7%	933.5%	960.2%
8192	434.0%	447.5%	460.9%	474.4%	487.8%	501.3%
16,384	225.6%	232.4%	239.2%	246.0%	252.8%	259.6%

APPENDIX B. MASPAR MPL CODES

B.1. Cannon's Matrix Multiplication Code

The files for the Cannon's Matrix Multiplication code are:

makefile - for the matrix multiplication

mmult.m - contains the main function that inputs the matrices, calling of the mx_mult() to perform the actual matrix multiplication, and output of the resulting matrix.

matrix.m - contains the function that actually computes the matrix multiplication

mtxio.m - contains the matrix I/O functions

dotprod.h - contains the macros to perform the dot product

matrix.h - contains macros for broadcasting to the diagonal, summing to the diagonal, and shifting the submatrices in all directions.

```
#########################################################################
# makefile for matrix multiplication
#########################################################################
SUFFIXES: .o .m .c
MPFLAGS = -Zq -Zn -nohprofile -Omax

mm: matrix.h dotprod.h matrix.S rmult.m
	mpl_cc $(MPFLAGS) matrix.S -o mm rmult.m;mplimit mm pmem 16k

io.o: io.m
	mpl_cc $(MPFLAGS) -c io.m

mtxio.o: mtxio.m
	mpl_cc $(MPFLAGS) -c mtxio.m
```

```
/*************************************************************************
File: mmult.m
Programmer: Jeff Clary (Modified by Mark Fienup)
*************************************************************************/
/*************************************************************************/
/* This file contains the main function that controls the input of   */
/* matrices to multiply, the output of the result, and the calling of */
/* mx_mult() to do the actual multiplication.                  */
/*************************************************************************/
#include "matrix.h"

main (argc, argv)
  int argc;
  char *argv[];
  {
  MATRIX A, B, C;

  double elapsed;
  FILE *afile, *bfile, *cfile;
  register int mlen1, mlen2;      /* Matrix length             */
/* for mp-1
mlen1 = 2048;
*/

/* for mp-2 */
mlen1 = 4096;

  if (open_files(&afile, &bfile, &cfile, argc, argv) < 0)
    exit(-1);

/* Read matrix A */
  dpuTimerStart();
  if ((mlen1 = mx_bread(afile, A)) < 0)
    exit(-1);
  elapsed = dpuTimerElapsed();

  printf("\n%d x %d MATRIX MULTIPLY ON %d x %d PE ARRAY\n\n",
       mlen1, mlen1, nxproc, nyproc);

  printf(" Reading A matrix: %10.4lf sec.\n", elapsed);

/* Read matrix B */
  dpuTimerStart();
```

```
  if ((mlen2 = mx_bread(bfile, B)) < 0)
    exit(-1);
  elapsed = dpuTimerElapsed();
  printf(" Reading B matrix: %10.4lf sec.\n", elapsed);

  if (mlen1 != mlen2)
    {
    printf("mmult: Matrix sizes (%d, %d) do not match\n", mlen1, mlen2);
    exit(-1);
    }

  /* Perform the matrix multiplication */
  if (mx_mult(A, B, C, mlen1) < 0)
    exit(-1);

/* Write the resulting matrix C */
  dpuTimerStart();
  mx_bwrite(cfile, C, mlen1);
  elapsed = dpuTimerElapsed();
  printf(" Writing C matrix: %10.4lf sec.\n", elapsed);

  fclose(afile);
  fclose(bfile);
  fclose(cfile);
*/
}
```

```
/*************************************************************************
File: matrix.m
Programmer: Jeff Clary (Modified by Mark Fienup)
*************************************************************************/
/*************************************************************************/
/* This file contains the function that does the actual matrix       */
/* multiplication.                                                   */
/*************************************************************************/
#include "matrix.h"
#include "dotprod.h"

/*-----------------------------------------------------------------*/
/* This function computes the matrix product C = A * B.            */
/* (Systolic Version -- Algorithm II)                              */
/* Note that the shifting of A and B necessary for the systolic alg. */
/* is performed by this function, so all three matrices are expected */
/* to be in normal position, i.e. block decomposed.  After the     */
/* routine returns, A and B will NOT be in that normal position.   */
/*-----------------------------------------------------------------*/
mx_mult(A, B, C, mlen)
  plural ELEM *A, *B, *C;
  int mlen;
  {
  register plural ELEM ctmp;
  register plural ELEM *arow, *bcol;
  register plural ELEM *cptr;

        double elapsed;
  register unsigned iter,i;
  register unsigned j;
  register unsigned blen = mlen / nxproc;
        unsigned bsize = blen * blen;


  /* ZERO OUT C MATRIX */
  for (j=bsize, cptr=C; j; j--, cptr++)
    *cptr = 0.0;

  /* SHIFT A SO THAT DIAGONAL ELEMENTS ARE AT RIGHT EDGE */
  for (j=nyproc; j>0; j--)
    {
    if (iyproc < j)
      shiftE(A,bsize);
```

```
  }

/* SHIFT B SO THAT DIAGONAL ELEMENTS ARE AT BOTTOM EDGE */
for (j=nxproc; j>0; j--)
  {
  if (ixproc < j)
    shiftS(B,bsize);
  }

dpuTimerStart();
/* ITERATE FOR LENGTH OF PE ARRAY */
for (iter=nxproc; iter; iter--)
  {
  /* EACH PE CALC  C=A*B  ON ITS SUBMATRIX */
  arow = A;
  cptr = C;
  for (i=blen; i; i--)
    {
    bcol = B;
    for (j=blen; j; j--)
      {
      dotprod(ctmp,arow,bcol,blen);
      *cptr += ctmp;
      bcol++;
      cptr++;
      }
    arow += blen;
    }
  /* SHIFT A,B ACCORDING TO SYSTOLIC ALGORITHM */
  shiftW(A,bsize);
  shiftN(B,bsize);
  }
elapsed = dpuTimerElapsed();
printf("%6d %10.4lf\n", mlen, elapsed);
return 0;
}
```

```
/*************************************************************************/
/* FILE: mtxio.m                                                         */
/*                                                                       */
/* This file contains routines for the input/output of matrices on       */
/* the MasPar, where the matrices are scatter or block decomposed.       */
/*************************************************************************/
#include <mpl.h>
#include <stdlib.h>
#include <stdio.h>

/*-----------------------------------------------------------------------*/
/* Function: mtx_alloc()   (Matrix allocation)                           */
/*                                                                       */
/*-----------------------------------------------------------------------*/
plural char *mtx_alloc(rows, cols, elemsize)
  unsigned rows, cols, elemsize;
  {
  unsigned brows = 0;
  unsigned bcols = 0;

  brows = rows/nyproc + (rows%nyproc ? 1 : 0);
  bcols = cols/nxproc + (cols%nxproc ? 1 : 0);

  return p_malloc(brows*bcols*elemsize);
  }

chk_decomp(row, col)
  char row, col;
  {
  if ((row != 'b' && row != 's') || (col != 'b' && col != 's'))
    {
    fprintf(stderr, "mtx_ardf: 'b' or 's' decomposition type required\n");
    return -1;
    }
  return 0;
  }

set_indices(ip, jp, offset, row_d, col_d, i, j, brows, bcols)
  unsigned *ip, *jp, *offset;
  unsigned i, j, brows, bcols;
  char row_d, col_d;
  {
  *ip = (row_d=='b') ? i/brows: i%nyproc;
```

```
  *jp = (col_d=='b') ? j/bcols: j%nxproc;
  *offset = ((row_d=='b') ? (i%brows)*bcols : (i/nyproc)*bcols)
        + ((col_d=='b') ? j%bcols : j/nxproc);
  }

/*-----------------------------------------------------------------*/
/* Function: mtx_brdf()    (Matrix binary read float)              */
/*                                                                 */
/* This function reads a binary array onto the MasPar array using  */
/* the decomposition scheme specified by row_decomp and col_decomp, */
/* where 's' means scatter and 'b' means block decomposition.       */
/*-----------------------------------------------------------------*/
mtx_brdf (fp, m, rows, cols, row_decomp, col_decomp, alloc_flag)
  FILE       *fp;
  plural float **m;
  unsigned    *rows;
  unsigned    *cols;
  char         row_decomp;
  char         col_decomp;
  int          alloc_flag;
  {
  unsigned brows, bcols;
  unsigned i, ip;
  unsigned j, jp;
  unsigned offset;
  float elem;

  if (chk_decomp(row_decomp, col_decomp) < 0)
    return -1;

  if (fread(rows, sizeof(*rows), 1, fp)!=1)
    {
    fprintf(stderr, "mtx_arbf: Error reading matrix rows\n");
    return -1;
    }
  if (fread(cols, sizeof(*cols), 1, fp)!=1)
    {
    fprintf(stderr, "mtx_arbf: Error reading matrix rows\n");
    return -1;
    }

  brows = *rows/nyproc;
  bcols = *cols/nxproc;
```

```
  if (alloc_flag)
    if ((*m=(plural float *)mtx_alloc(*rows,*cols,sizeof(float)))==NULL)
      return -1;

  for (i=0; i<*rows; i++)
    {
    for (j=0; j<*cols; j++)
      {
      set_indices(&ip, &jp, &offset, row_decomp, col_decomp,
              i, j, brows, bcols);
      if (fread(&elem, sizeof(elem), 1, fp)!=1)
        {
        fprintf(stderr, "mtx_ardf: Error reading elem %d, %d\n", i,j);
        return -1;
        }
      proc[ip][jp].((*m)[offset]) = elem;
      }
    }
  return 0;
  }

/*-----------------------------------------------------------------*/
/* Function: mtx_bwtf()    (Matrix binary write float)             */
/*                                                                 */
/* This function writes a binary array from the MasPar array using  */
/* the decomposition scheme specified by row_decomp and col_decomp,  */
/* where 's' means scatter and 'b' means block decomposition.       */
/*-----------------------------------------------------------------*/
mtx_bwtf (fp, m, rows, cols, row_decomp, col_decomp)
  FILE        *fp;
  plural float *m;
  unsigned     rows;
  unsigned     cols;
  char         row_decomp;
  char         col_decomp;
  {
  unsigned brows, bcols;
  unsigned i, ip;
  unsigned j, jp;
  unsigned offset;
  float elem;
```

```
brows = rows/nyproc;
bcols = cols/nxproc;

if (chk_decomp(row_decomp, col_decomp) < 0)
  return -1;

if (fwrite(&rows, sizeof(rows), 1, fp)!=1)
  {
  fprintf(stderr, "mtx_bwtf: error writing rows\n");
  return -1;
  }
if (fwrite(&cols, sizeof(rows), 1, fp)!=1)
  {
  fprintf(stderr, "mtx_bwtf: error writing cols\n");
  return -1;
  }

for (i=0; i<rows; i++)
  {
  for (j=0; j<cols; j++)
    {
    set_indices(&ip, &jp, &offset, row_decomp, col_decomp,
            i, j, brows, bcols);
    elem = proc[ip][jp].(m[offset]);
    if (fwrite(&elem, sizeof(elem), 1, fp)!=1)
      {
      fprintf(stderr, "mtx_bwtf: error writing elem (%d,%d)\n",i,j);
      return -1;
      }
    }
  }
return 0;
}
```

```
/*************************************************************************
File: matrix.h
*************************************************************************/
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpl.h>

void    dpuTimerStart();
unsigned dpuTimerTicks();
double  dpuTimerConst();
double  dpuTimerElapsed();

#define ELEMFMT "%f "
#define ELEM float

/*************************************************************************/
/* For running maximum size problems on the MP-1, BUFSIZE should be  */
/* 1024.  For running maximum size problems on the MP-2, BUFSIZE     */
/* should be 4096.  (Except Alg3 must be compiled with BUFSIZE less) */
/*************************************************************************/

/* Defined to get MP-2 declarations */
#define JC_MP2
/*
#ifdef JC_MP2
#define BUFSIZE  4096
#define BUFSIZE2 3136
#else
#define BUFSIZE  1024
#define BUFSIZE2  576
#endif
*/

#define BUFSIZE  4096
#define BUFSIZE2 3136

typedef plural ELEM MATRIX[BUFSIZE];
/*
#ifdef _MPL
typedef plural ELEM MATRIX[BUFSIZE];
#else
typedef ELEM MATRIX[BUFSIZE];
```

```
#endif
*/

double stopwatch();
int   open_files();
int   mx_bread();
int   mx_bwrite();
int   mx_mult();

#define broaddiag(Xto, Xfrom, size, mask)              \
  {                                                    \
  register plural ELEM x1, x2, x3;                     \
  register plural ELEM *from, *to;                     \
  register int i;                                      \
  from = Xfrom;                                        \
  to   = Xto;                                          \
  x1 = *from++;                                        \
  for (i=(size-1)>>1; i; i--)                          \
    {                                                  \
    x2 = *from++;                                      \
    if (mask) xnetcE[nxproc].x1 = x1;                  \
    x3 = *from++;                                      \
    *to++ = x1;                                        \
    if (mask) xnetcE[nxproc].x2 = x2;                  \
    *to++ = x2;                                        \
    x1 = x3;                                           \
    }                                                  \
  if (!(size & 0x01))                                  \
    {                                                  \
    x2 = *from;                                        \
    if (mask) xnetcE[nxproc].x1 = x1;                  \
    *to++ = x1;                                        \
    if (mask) xnetcE[nxproc].x2 = x2;                  \
    *to = x2;                                          \
    }                                                  \
  else                                                 \
    {                                                  \
    if (mask) xnetcE[nxproc].x1 = x1;                  \
    *to = x1;                                          \
    }                                                  \
  }
```

```
/*-------------------------------------------------------------*/
/* Macro: sum_to_diag(term,diag,i,diagidx)                     */
/*                                                             */
/* The macro sums into the column with offidx==0               */
/*                                                             */
/* plural unsigned diagidx = (nxproc+ixproc-iyproc)%nxproc;    */
/* must be supplied by the caller but is not modified.         */
/*                                                             */
/* This macro requires all PE's active on entry.               */
/*-------------------------------------------------------------*/
#define sum_to_diag(term,offidx)                    \
  {                                                 \
  register int i;                                   \
  for (i=1; i<nxproc; i<<=1)                        \
    if (offidx & i)                                 \
      xnetpW[i].term += term;                       \
  }

/*-------------------------------------------------------------*/
/* Macro: nc_sum_to_diag(term,diag,i,diagidx)                  */
/*                                                             */
/* This macro simulates the non-communication functions of     */
/* sum_to_diag.                                                */
/*-------------------------------------------------------------*/
#define nc_sum_to_diag(term,offidx)                 \
  {                                                 \
  register int i;                                   \
  for (i=1; i<nxproc; i<<=1)                        \
    if (offidx & i)                                 \
     /*  xnetpW[i].term += term; */                 \
        term += term;                               \
  }
/*-------------------------------------------------------------*/
/* These macros shift a MATRIX (a block of ELEM's) one PE in the */
/* specified direction.  size indicates the number of elements in the */
/* block.                                                      */
/*                                                             */
/* The scratch variables                                       */
/*   MATRIX from, to;                                          */
/*   int i;                                                    */
/* must be supplied by the caller and will be modified.  Presumably, */
/* the caller will supply register variables.                  */
```

```
/*-----------------------------------------------------------------*/
#define shiftN(X,size)              \
  {                                 \
  register plural ELEM x1, x2, x3;    \
  register plural ELEM *from, *to;          \
  register int i;                   \
  from = to = X;                      \
  x1 = *from++;                       \
  for (i=(size-1)>>1; i; i--)         \
    {                               \
    x2 = *from++;                     \
    xnetN[1].x1 = x1;                 \
    x3 = *from++;                     \
    *to++ = x1;                     \
    xnetN[1].x2 = x2;                 \
    *to++ = x2;                     \
    x1 = x3;                      \
    }                               \
  if (!(size & 0x01))                 \
    {                               \
    x2 = *from;                     \
    xnetN[1].x1 = x1;                 \
    *to++ = x1;                     \
    xnetN[1].x2 = x2;                 \
    *to = x2;                     \
    }                               \
  else                              \
    {                               \
    xnetN[1].x1 = x1;                 \
    *to = x1;                     \
    }                               \
  }

#define shiftS(X,size)              \
  {                                 \
  register plural ELEM x1, x2, x3;    \
  register plural ELEM *from, *to;          \
  register int i;                   \
  from = to = X;                      \
  x1 = *from++;                       \
  for (i=(size-1)>>1; i; i--)         \
    {                               \
    x2 = *from++;                     \
```

```
    xnetS[1].x1 = x1;              \
    x3 = *from++;                  \
    *to++ = x1;                   \
    xnetS[1].x2 = x2;              \
    *to++ = x2;                   \
    x1 = x3;                     \
    }                            \
  if (!(size & 0x01))              \
    {                            \
    x2 = *from;                   \
    xnetS[1].x1 = x1;              \
    *to++ = x1;                   \
    xnetS[1].x2 = x2;              \
    *to = x2;                    \
    }                            \
  else                           \
    {                            \
    xnetS[1].x1 = x1;              \
    *to = x1;                    \
    }                            \
  }

#define shiftE(X,size)             \
  {                              \
  register plural ELEM x1, x2, x3;    \
  register plural ELEM *from, *to;          \
  register int i;                \
  from = to = X;                  \
  x1 = *from++;                  \
  for (i=(size-1)>>1; i; i--)       \
    {                            \
    x2 = *from++;                  \
    xnetE[1].x1 = x1;              \
    x3 = *from++;                  \
    *to++ = x1;                   \
    xnetE[1].x2 = x2;              \
    *to++ = x2;                   \
    x1 = x3;                     \
    }                            \
  if (!(size & 0x01))              \
    {                            \
    x2 = *from;                   \
    xnetE[1].x1 = x1;              \
```

```
    *to++ = x1;                \
    xnetE[1].x2 = x2;          \
    *to = x2;                  \
    }                          \
  else                         \
    {                          \
    xnetE[1].x1 = x1;          \
    *to = x1;                  \
    }                          \
  }

#define shiftW(X,size)         \
  {                            \
  register plural ELEM x1, x2, x3;    \
  register plural ELEM *from, *to;    \
  register int i;              \
  from = to = X;               \
  x1 = *from++;                \
  for (i=(size-1)>>1; i; i--)  \
    {                          \
    x2 = *from++;              \
    xnetW[1].x1 = x1;          \
    x3 = *from++;              \
    *to++ = x1;                \
    xnetW[1].x2 = x2;          \
    *to++ = x2;                \
    x1 = x3;                   \
    }                          \
  if (!(size & 0x01))          \
    {                          \
    x2 = *from;                \
    xnetW[1].x1 = x1;          \
    *to++ = x1;                \
    xnetW[1].x2 = x2;          \
    *to = x2;                  \
    }                          \
  else                         \
    {                          \
    xnetW[1].x1 = x1;          \
    *to = x1;                  \
    }                          \
  }
```

```
/*************************************************************************
                dotprod.h
*************************************************************************/
/*************************************************************************/
/* This file contains macros for calculating the dot product of      */
/* vectors A and B (of length blen) into c.                          */
/*************************************************************************/

/* Vanilla version -- no special optimization */
#define dotprod(c,A,B,blen)        \
  {                                \
  register plural ELEM *aptr, *bptr;     \
  register int i;                  \
  aptr = A;                        \
  bptr = B;                        \
  c = 0.0;                         \
  for (i=blen; i; i--)             \
    {                              \
    c += (*aptr) * (*bptr);        \
    aptr++;                        \
    bptr += blen;                  \
    }                              \
  }

/* First crack at memory overlap optimization */
#define dotprod1(c,A,B,blen)             \
  {                                      \
  register plural ELEM *aptr, *bptr;           \
  register plural ELEM a1, a2, b1, b2;   \
  register int i;                        \
  aptr = A;                              \
  a1   = *aptr;                          \
  bptr = B;                              \
  b1   = *bptr;                          \
  c = 0.0;                               \
  for (i=(blen-1); i; i--)               \
    {                                    \
    aptr++;                              \
    a2 = *aptr;                          \
    bptr += blen;                        \
    b2 = *bptr;                          \
    c += a1 * b1;                        \
    a1 = a2;                             \
```

```
    b1 = b2;                    \
    }                           \
  c += a1 * b1;                 \
  }

/* Second crack at memory overlap optimization -- depth 2 unroll */
/* NOTE that this routine assumes blen is even.                  */
#define dotprod2(c,A,B,blen)            \
  {                                     \
  register plural ELEM *aptr, *bptr;            \
  register plural ELEM a1, a2, a3, b1, b2, b3;  \
  register int i;                       \
  aptr = A;                             \
  bptr = B;                             \
  a1   = *aptr;                         \
  b1   = *bptr;                         \
  aptr++;                               \
  bptr += blen;                         \
  c    = 0.0;                           \
  for (i=(blen-1)>>1; i; i--)           \
    {                                   \
    a2 = *aptr;                         \
    b2 = *bptr;                         \
    aptr++;                             \
    bptr += blen;                       \
    c += a1 * b1;                       \
    a3 = *aptr;                         \
    b3 = *bptr;                         \
    aptr++;                             \
    bptr += blen;                       \
    c += a2 * b2;                       \
    a1 = a3;                            \
    b1 = b3;                            \
    }                                   \
  a2 = *aptr;                           \
  b2 = *bptr;                           \
  c += a1 * b1;                         \
  c += a2 * b2;                         \
  }

/* Third crack at memory overlap optimization -- depth 4 unroll */
/* NOTE that this routine assumes blen is divisible by 4.       */
#define dotprod3(c,A,B,blen)            \
```

```
{                                   \
register plural ELEM *aptr, *bptr;          \
register plural ELEM a1, a2, a3, a4;        \
register plural ELEM b1, b2, b3, b4;        \
register int i;                             \
aptr = A;                                    \
bptr = B;                                    \
a1   = *aptr;                                \
b1   = *bptr;                                \
aptr++;                                     \
bptr += blen;                                \
a2 = *aptr;                                 \
b2 = *bptr;                                 \
aptr++;                                     \
bptr += blen;                                \
c    = 0.0;                                  \
for (i=(blen-1)>>2; i; i--)                       \
  {                                         \
  a3 = *aptr;                                \
  b3 = *bptr;                                \
  aptr++;                                   \
  bptr += blen;                              \
  a4 = *aptr;                               \
  b4 = *bptr;                               \
  aptr++;                                   \
  bptr += blen;                              \
  c += a1 * b1;                              \
  c += a2 * b2;                              \
  a1   = *aptr;                              \
  b1   = *bptr;                              \
  aptr++;                                   \
  bptr += blen;                              \
  a2 = *aptr;                               \
  b2 = *bptr;                               \
  aptr++;                                   \
  bptr += blen;                              \
  c += a3 * b3;                              \
  c += a4 * b4;                              \
  }                                         \
a3 = *aptr;                                 \
b3 = *bptr;                                 \
aptr++;                                     \
bptr += blen;                                \
```

```
  a4 = *aptr;                          \
  b4 = *bptr;                          \
  c += a1 * b1;                        \
  c += a2 * b2;                        \
  c += a3 * b3;                        \
  c += a4 * b4;                        \
  }

/* This version of dotprod moves each A element to the west */
/* after it is used                                         */
#define dotprod3A(c,A,B,blen)                \
  {                                  \
  register plural ELEM *aptr, *bptr;         \
  register plural ELEM a1, a2, a3, a4;       \
  register plural ELEM b1, b2, b3, b4;       \
  register int i;                    \
  aptr = A;                          \
  bptr = B;                          \
  a1   = *aptr;                      \
  b1   = *bptr;                      \
  aptr++;                           \
  bptr += blen;                       \
  a2 = *aptr;                        \
  b2 = *bptr;                        \
  aptr++;                           \
  bptr += blen;                       \
  c    = 0.0;                       \
  for (i=(blen-1)>>2; i; i--)              \
    {                               \
    a3 = *aptr;                       \
    b3 = *bptr;                       \
    aptr++;                          \
    bptr += blen;                      \
    a4 = *aptr;                       \
    b4 = *bptr;                       \
    aptr++;                          \
    bptr += blen;                      \
    c += a1 * b1;                     \
    c += a2 * b2;                     \
    xnetW[1].a1 = a1;                    \
    *(aptr-4) = a1;                      \
    xnetW[1].a2 = a2;                    \
    *(aptr-3) = a2;                      \
```

```
    a1 = *aptr;                    \
    b1 = *bptr;                    \
    aptr++;                       \
    bptr += blen;                  \
    a2 = *aptr;                    \
    b2 = *bptr;                    \
    aptr++;                       \
    bptr += blen;                  \
    c += a3 * b3;                  \
    c += a4 * b4;                  \
    xnetW[1].a3 = a3;                \
    *(aptr-4) = a3;                \
    xnetW[1].a4 = a4;                \
    *(aptr-3) = a4;                \
    }                            \
  a3 = *aptr;                      \
  b3 = *bptr;                      \
  aptr++;                         \
  bptr += blen;                    \
  a4 = *aptr;                      \
  b4 = *bptr;                      \
  c += a1 * b1;                    \
  c += a2 * b2;                    \
  c += a3 * b3;                    \
  c += a4 * b4;                    \
  xnetW[1].a1 = a1;                \
  *(aptr-3) = a1;                  \
  xnetW[1].a2 = a2;                \
  *(aptr-2) = a2;                  \
  xnetW[1].a3 = a3;                \
  *(aptr-1) = a3;                  \
  xnetW[1].a4 = a4;                \
  *(aptr) = a4;                   \
  }

/* This version of dotprod moves each B element to the north */
/* after it is used                                          */
#define dotprod3B(c,A,B,blen)          \
  {                              \
  register plural ELEM *aptr, *bptr;     \
  register plural ELEM a1, a2, a3, a4;   \
  register plural ELEM b1,b2,b3,b4;      \
```

```
register int blen4=blen<<2,blen3=(blen<<1)+blen;\
register int i;                          \
aptr = A;                                \
bptr = B;                                \
a1   = *aptr;                            \
b1   = *bptr;                            \
aptr++;                                  \
bptr += blen;                             \
a2 = *aptr;                              \
b2 = *bptr;                              \
aptr++;                                  \
bptr += blen;                             \
c    = 0.0;                              \
for (i=(blen-1)>>2; i; i--)                   \
  {                                    \
  a3 = *aptr;                             \
  b3 = *bptr;                             \
  aptr++;                                 \
  bptr += blen;                            \
  a4 = *aptr;                             \
  b4 = *bptr;                             \
  aptr++;                                 \
  bptr += blen;                            \
  c += a1 * b1;                           \
  c += a2 * b2;                           \
  xnetN[1].b1 = b1;                          \
  *(bptr-blen4) = b1;                              \
  xnetN[1].b2 = b2;                          \
  *(bptr-blen3) = b2;                              \
  a1 = *aptr;                             \
  b1 = *bptr;                             \
  aptr++;                                 \
  bptr += blen;                            \
  a2 = *aptr;                             \
  b2 = *bptr;                             \
  aptr++;                                 \
  bptr += blen;                            \
  c += a3 * b3;                           \
  c += a4 * b4;                           \
  xnetN[1].b3 = b3;                          \
  *(bptr-blen4) = b3;                             \
  xnetN[1].b4 = b4;                          \
  *(bptr-blen3) = b4;                             \
```

```
  }                                \
  a3 = *aptr;                       \
  b3 = *bptr;                       \
  aptr++;                          \
  bptr += blen;                     \
  a4 = *aptr;                      \
  b4 = *bptr;                      \
  c += a1 * b1;                     \
  c += a2 * b2;                     \
  c += a3 * b3;                     \
  c += a4 * b4;                     \
  xnetN[1].b1 = b1;                   \
  *(bptr-blen3) = b1;                \
  xnetN[1].b2 = b2;                  \
  *(bptr-(blen<<1)) = b2;              \
  xnetN[1].b3 = b3;                  \
  *(bptr-(blen)) = b3;               \
  xnetN[1].b4 = b4;                  \
  *(bptr) = b4;                    \
  }
```

```
/* This version of dotprod moves each B element to the north */
/* and each A element to the west after it is used.        */
#define dotprod3AB(c,A,B,blen)              \
  {                                \
  register plural ELEM *aptr, *bptr;          \
  register plural ELEM a1, a2, a3, a4;        \
  register plural ELEM b1,b2,b3,b4;           \
  register int blen4=blen<<2,blen3=(blen<<1)+blen;\
  register int i;                  \
  aptr = A;                        \
  bptr = B;                        \
  a1   = *aptr;                    \
  b1   = *bptr;                    \
  aptr++;                          \
  bptr += blen;                     \
  a2 = *aptr;                     \
  b2 = *bptr;                     \
  aptr++;                         \
  bptr += blen;                     \
  c    = 0.0;                      \
```

```
for (i=(blen-1)>>2; i; i--)                \
  {                                        \
  a3 = *aptr;                               \
  b3 = *bptr;                               \
  aptr++;                                   \
  bptr += blen;                             \
  a4 = *aptr;                               \
  b4 = *bptr;                               \
  aptr++;                                   \
  bptr += blen;                             \
  c += a1 * b1;                             \
  c += a2 * b2;                             \
  xnetN[1].b1 = b1;                         \
  *(bptr-blen4) = b1;                             \
  xnetN[1].b2 = b2;                         \
  *(bptr-blen3) = b2;                             \
  xnetW[1].a1 = a1;                         \
  *(aptr-4) = a1;                           \
  xnetW[1].a2 = a2;                         \
  *(aptr-3) = a2;                           \
  a1 = *aptr;                               \
  b1 = *bptr;                               \
  aptr++;                                   \
  bptr += blen;                             \
  a2 = *aptr;                               \
  b2 = *bptr;                               \
  aptr++;                                   \
  bptr += blen;                             \
  c += a3 * b3;                             \
  c += a4 * b4;                             \
  xnetN[1].b3 = b3;                         \
  *(bptr-blen4) = b3;                             \
  xnetN[1].b4 = b4;                         \
  *(bptr-blen3) = b4;                             \
  xnetW[1].a3 = a3;                         \
  *(aptr-4) = a3;                           \
  xnetW[1].a4 = a4;                         \
  *(aptr-3) = a4;                           \
  }                                        \
a3 = *aptr;                                 \
b3 = *bptr;                                 \
aptr++;                                     \
bptr += blen;                               \
```

```
a4 = *aptr;                    \
b4 = *bptr;                    \
c += a1 * b1;                  \
c += a2 * b2;                  \
c += a3 * b3;                  \
c += a4 * b4;                  \
xnetN[1].b1 = b1;              \
xnetN[1].b2 = b2;              \
xnetN[1].b3 = b3;              \
xnetN[1].b4 = b4;              \
*(bptr-blen3) = b1;                \
*(bptr-(blen<<1)) = b2;                \
*(bptr-blen) = b3;                 \
*(bptr) = b4;                  \
xnetW[1].a1 = a1;              \
xnetW[1].a2 = a2;              \
xnetW[1].a3 = a3;              \
xnetW[1].a4 = a4;              \
*(aptr-3) = a1;                \
*(aptr-2) = a2;                \
*(aptr-1) = a3;                \
*(aptr) = a4;                  \
}
```

B.2. Gauss-Jordan Elimination Code

The files for the Gauss-Jordan Elimination (with partial pivoting) program are:

makefile - contains a make file to compile the GJE code

linSysSolv.h - contains the necessary includes and function prototypes

main.m - contains the main function that controls the input of matrix A and vector b, the output of the result, and the calling of linSysSolver() to do the actual solving of Ax = b.

linsolv.m - contains the function that actually solves the linear equations using Gauss-Jordan Elimination with partial pivoting. In this code the pivot rows are actually swapped.

```
#####################################################################
# makefile
#####################################################################
.SUFFIXES: .o .m .c
MPFLAGS = -Zq -Zn -nohprofile -Omax

all:  linSolv kim random mdiff atob btoa

linSolv: linSysSolver.h linSolv.o main.m
        mpl_cc $(MPFLAGS) linSolv.o -o linSolv main.m -lmpml;mplimit linSolv pmem 16k

io.o: linSysSolver.h io.m
        mpl_cc $(MPFLAGS) -c io.m

mtxio.o: mtxio.m
        mpl_cc $(MPFLAGS) -c mtxio.m

linSolv.o: linSysSolver.h linSolv.m
        mpl_cc $(MPFLAGS) mtxio.o -c linSolv.m
```

```
/*************************************************************************
File: linSysSolver.h
Programmer: Mark Fienup
*************************************************************************/

#include <mpl.h>
#include <mpml.h>
#include <math.h>
#include <reduce.h>
#include <stdio.h>

#define ETYPE  float

plural char  *p_malloc();
char         *malloc();
void         perror();
void         free();
void         p_free();
double       linSysSolver();
int          open_files();
int          mx_bread();
int          mx_bwrite();
int          exit();
int          atoi();

void dpuTimerStart();
unsigned long dpuTimerTicks();
double dpuTimerConst();
double  dpuTimerElapsed();
```

```
/*************************************************************************
File: main.m
Programmer: Mark Fienup
*************************************************************************/
/*************************************************************************/
/* This file contains the main function that controls the input of   */
/* matrix A and vector b, the output of the result, and the calling of */
/* linSysSolver() to do the actual solving of Ax = b.              */
/*************************************************************************/
#include "linSysSolver.h"

main (argc, argv)
    int argc;
    char *argv[];
{
 plural ETYPE *a, *x, *b, *tmpptr;
 int bcols, brows;


 double elapsed;
 register int n, c, i;       /* Matrix length */

/* for full range of beta start i at 4 to 64 by 4 */
 for (i=1; i < 4; i += 1) {
  n = i*nxproc;
  bcols = ((n - 1)>>lnxproc) + 1;
  brows = ((n - 1)>>lnyproc) + 1;

/* Allocate large enough local arrays */
   if ((a = (plural ETYPE *) p_malloc(brows*bcols * sizeof(ETYPE))) == NULL) {
    perror("memory allocation error:  a");
    return -1;
   }

   if ((b = (plural ETYPE *) p_malloc(brows * sizeof(ETYPE))) == NULL) {
    perror("memory allocation error: b");
    return -1;
   }

/* Fills the matrices with random data */
  tmpptr = a;
  for (i = 0; i < brows*bcols; i++) {
    fp_matran(nyproc, nxproc, tmpptr, nxproc, 0, 0);
```

```
    tmpptr++;
  }

  tmpptr = b;
  for (i = 0; i < brows; i++) {
    fp_matran(nyproc, nxproc, tmpptr, nxproc, 0, 0);
    tmpptr++;
  }

  dpuTimerStart();

/* Solve the linear system of equations */
  if (linSysSolver(n, a, x, b) < 0)
    exit(-1);
  elapsed = dpuTimerElapsed();

/* Print the timing information */
  printf("Beta= %8d  n = %6d  nxproc = %6d  Time = %10.5lf sec.\n",
        i*i, n, nxproc, elapsed);

  p_free(a);
  p_free(b);
 }
```

```
/*****************************************************************************
File: linsolv.m
Programmer: Mark Fienup
*****************************************************************************/
/*****************************************************************************
 Procedure: linSysSolver

 This procedure solves the linear system of equations Ax = b. This
 is accomplished by performing pivoting, which actually swaps the rows.
 All off diagonal elements are zeroed and the diagonal elements are made
 to be one.

 Input:
   n: the length of A (i.e., A is an n x n matrix)
   A: an n x n matrix which is scatter decomposed onto the PEs
   b: a vector of length n which is 1d scatter decomposed onto column 0

 Output:
   x: the solution vector
*****************************************************************************/
#include "linSysSolver.h"

#define max_to_row_0(temp,temp2,temp3,temp4)          \
  {                                                   \
  register int i;                                     \
  for (i=1; i<nxproc; i<<=1)                          \
        if (iyproc & i) {                             \
          xnetpS[i].temp2 = temp;                     \
          xnetpS[i].temp4 = temp3;                    \
          }                                           \
        if (temp2 > temp) {                           \
          temp = temp2;                               \
          temp3 = temp4;                              \
          }                                           \
  }


double linSysSolver(int n, plural ETYPE a[], plural ETYPE x[],
        plural ETYPE b[])
{
 register plural ETYPE *arow;
 register plural ETYPE mult;
 register plural ETYPE *tmpptr, *tmpptr2, *tmpptr3, *tmpptr4, *rowptr;
```

```
register plural ETYPE *curptr, *cur_pos, *cur_pos1;
register plural ETYPE scale;
register int x_layer, y_layer, x_index, y_index, nxproc_1=nxproc - 1;

/* register variables for depth 2 loop unrolling */
register int col_loops, col_loops2;

register int maxdist, maxpost, loc;
register ETYPE temp;

register plural int maxloc, Loc1, Loc2;
register plural ETYPE maxval, *maxptr;

register plural ETYPE R0, R1, R2, R3, R4, *PR0;

register int bcols, brows, half_nyproc = nyproc>>1;
register int i, j, k, rr, cc;
register int  rem;
int debug;

debug = 0;

bcols = ((n - 1)>>lnxproc) + 1;
brows = ((n - 1)>>lnyproc) + 1;

/* Allocate a local row that's used when exchanging the pivot rows */
 if ((arow = (plural ETYPE *) p_malloc(bcols * sizeof(ETYPE))) == NULL) {
  perror("memory allocation error");
  return -1;
 }

 /* pad remainder of block with zero if n is not mutiple of nxproc/nyproc */
 if (rem = (n - (n>>lnxproc)*nxproc)) {
  if (ixproc >= rem) {
   tmpptr = a + bcols - 1;
   for (i = 0; i < bcols; i++) {
       *tmpptr = 0.0;
       tmpptr += bcols;
    }
  }
 }
 if (rem = (n - (n>>lnyproc)*nyproc)) {
  if (iyproc >= rem) {
```

```
    tmpptr = a + bcols*(brows-1);
    for (i = 0; i < brows; i++) {
        *tmpptr = 0.0;
        tmpptr++;
    }
  }
}

curptr = a + bcols - 1;     /* ptr to 1st elt in last column of a PE */
cur_pos = a - bcols - 1;    /* ptr to pivot elt?*/

/* for each column do - main loop*/
for (i = 0; i < n; i++) {

  x_layer = i>>lnxproc;              /* col. layer of i */
  y_layer = i>>lnyproc;              /* row. layer of i */
  x_index = i - (x_layer<<lnxproc);  /* ixproc of pivot PE */
  y_index = i - (y_layer<<lnyproc);  /* iyproc of pivot PE */

  col_loops = bcols-x_layer-1;       /* no. of col. layer to rt. of pivot
                                        layer */

  col_loops2 = col_loops>>1;         /* half as much used for loop unrolling*/

  if (!x_index)              /* if pivot PE has ixproc = 0 */
    cur_pos += bcols + 1;

  if (col_loops%2) {  /* if the # of remaining column layers is odd then */

    /* find maximal pivot - divide and conquer */

    /* find the maximal pivot on each PE in the column first */
    maxval = -1.0;
    if (ixproc == x_index) {
        /* tmpptr set to ptr. to the last elt. in column */
        tmpptr = cur_pos + (brows-y_layer-1)*bcols;

        /* software pipelined */
        R0 = *tmpptr;

        for (j = brows-1; j > y_layer; j--) {
          tmpptr -= bcols;
          R1 = *tmpptr;
```

```
      if (R0 < 0) R0 = -R0;
      if (R0 > maxval) {
        maxval = R0;
        maxloc = j;
      }
      R0 = R1;
    }
    if (R0 < 0) R0 = -R0;
    if (iyproc >= y_index) {
      if (R0 > maxval) {
        maxval = R0;
        maxloc = j;
      }
    }

    Loc1 = iyproc;
    /* find the maximal pivot on all PEs in the pivot column */
    max_to_row_0(maxval,R0,Loc1,Loc2);
    /* loc is the iyproc of max. elt */
    loc = proc[0][x_index].Loc1;
}


 /* maxpost is y_layer in which the max. pivot elt found */
    maxpost = proc[loc][x_index].maxloc;

/* maxptr is set to the last elt in row where pivot elt found */
maxptr = curptr + maxpost*bcols;
/* tmpptr2 ptrs to last elt in arow so a copy of the pivot row can be
     stored starting at the front of arow */
tmpptr2 = arow + bcols - 1 - x_layer;

/* try to swap the b values of the swapped rows */
/* tmpptr3 ptrs to the elt in the b vector corresponding to the row where
     the max. pivot elt was found */
tmpptr3 = b + maxpost;
/* tmpptr4 ptrs to the elt in the b vector corresponding to row i */
tmpptr4 = b + y_layer;

temp = proc[loc][loc].*tmpptr3;
proc[loc][loc].*tmpptr3 = proc[y_index][y_index].*tmpptr4;
proc[y_index][y_index].*tmpptr4 = temp;
```

```
tmpptr = a;

/*
 * broadcast the pivot row below to the rows
 * all values of arow that represent ELEMS to the left
 * of the current pivot remain 0.0
 *
 * interchange rows
 */
if (iyproc == loc) { /* if PE in the row containing the max. pivot elt */
    if (maxdist = (loc - y_index)) { /* if rows are on different PEs */
      /* maxdist is the distance in rows for which the swap must
         be performed  */
      /* tmpptr set to pt to the last elt in ith row */
      tmpptr = curptr + y_layer*bcols;
      /* PR0 pts to the last elt in the row of the max. pivot elt */
      PR0 = maxptr;

      /* Determine the shortest direction for swapping the rows */
      if (maxdist > half_nyproc) maxdist -= nyproc;
      else if (maxdist < -half_nyproc) maxdist += nyproc;

      if (maxdist < 0) { /* if ith row PE "above" where max. pivot found */
        maxdist = -maxdist;

        /* swap rows: software pipelined and loop unrolled to a
           depth of 2 */
        R0 = *maxptr;
        for (j = 0; j < col_loops2; j++) {
          R1 = *--PR0;
          xnetcS[nyproc].R2 = R0;
          all *tmpptr2-- = R2;
          R4 = xnetpS[maxdist].*tmpptr;
          *maxptr = R4;
          all if (iyproc == y_index) *tmpptr-- = R2;
          maxptr = PR0;

          R3 = *--PR0;
          xnetcS[nyproc].R2 = R1;
          all *tmpptr2-- = R2;
          R4 = xnetpS[maxdist].*tmpptr;
          *maxptr = R4;
          all if (iyproc == y_index) *tmpptr-- = R2;
```

```
  maxptr = PR0;
  R0 = R3;
 }

 R1 = *--PR0;
 xnetcS[nyproc].R2 = R0;
 all *tmpptr2-- = R2;
 R4 = xnetpS[maxdist].*tmpptr;
 *maxptr = R4;
 all if (iyproc == y_index) *tmpptr-- = R2;

 maxptr = PR0;

 R4 = xnetpS[maxdist].*tmpptr;
 *maxptr = R4;
 xnetcS[nyproc].R2 = R1;
 maxdist = -maxdist;
}
else { /* if ith row "below" where the max. pivot row found */
 R0 = *maxptr;
 for (j = 0; j < col_loops2; j++) {
  R1 = *--PR0;
  xnetcS[nyproc].R2 = R0;
  all *tmpptr2-- = R2;
  R4 = xnetpN[maxdist].*tmpptr;
  *maxptr = R4;
  all if (iyproc == y_index) *tmpptr-- = R2;
  maxptr = PR0;
  R3 = *--PR0;
  xnetcS[nyproc].R2 = R1;
  all *tmpptr2-- = R2;
  R4 = xnetpN[maxdist].*tmpptr;
  *maxptr = R4;
  all if (iyproc == y_index) *tmpptr-- = R2;
  maxptr = PR0;
  R0 = R3;
 }
 R1 = *--PR0;
 xnetcS[nyproc].R2 = R0;
 all *tmpptr2-- = R2;
 R4 = xnetpN[maxdist].*tmpptr;
 *maxptr = R4;
 all if (iyproc == y_index) *tmpptr-- = R2;
```

```
    maxptr = PR0;
    R4 = xnetpN[maxdist].*tmpptr;
    *maxptr = R4;
    xnetcS[nyproc].R2 = R1;
   }
  }
  else { /* else the rows to be swapped are on the same PEs */
   tmpptr = curptr + y_layer*bcols;
   PR0 = maxptr;

   if (tmpptr == maxptr) { /* if no swapped of rows necessary then
                              only broadcast pivot row */
    R0 = *maxptr;
    for (j = 0; j < col_loops2; j++) {
     R1 = *--PR0;
     xnetcS[nyproc].R2 = R0;
     all *tmpptr2-- = R2;
     *tmpptr-- = R2;
     maxptr = PR0;
     R3 = *--PR0;
     xnetcS[nyproc].R2 = R1;
     all *tmpptr2-- = R2;
     *tmpptr-- = R2;
     maxptr = PR0;
     R0 = R3;
    }
    R1 = *--PR0;
    xnetcS[nyproc].R2 = R0;
    all *tmpptr2-- = R2;
    *tmpptr-- = R2;
    maxptr = PR0;
    xnetcS[nyproc].R2 = R1;
   }
   else {   /* swap rows on the same PEs and broadcast pivot row */

    R0 = *maxptr;
    for (j = 0; j < col_loops2; j++) {
     R1 = *--PR0;
     xnetcS[nyproc].R2 = R0;
     all *tmpptr2-- = R2;
     *maxptr = *tmpptr;
     *tmpptr-- = R2;
     maxptr = PR0;
```

```
            R3 = *--PR0;
            xnetcS[nyproc].R2 = R1;
            all *tmpptr2-- = R2;
            *maxptr = *tmpptr;
            *tmpptr-- = R2;
            maxptr = PR0;
            R0 = R3;
           }
          R1 = *--PR0;
          xnetcS[nyproc].R2 = R0;
          all *tmpptr2-- = R2;
          *maxptr = *tmpptr;
          *tmpptr-- = R2;
          maxptr = PR0;
          *maxptr = *tmpptr;
          xnetcS[nyproc].R2 = R1;
         }
        }
}

if (ixproc > x_index) {
    *tmpptr2 = R2;
}
else {
    *tmpptr2 = 0.0;
}

if (iyproc == y_index) {
    *tmpptr-- = R2;
}

if (ixproc == x_index)
    scale = 1.0 / proc[y_index][x_index].R2;

/*
 * for each subrow
 */

tmpptr3 = b;

/* Updates rows in the layers above the y_layer (current layer) */
cur_pos1 = cur_pos - y_layer*bcols - bcols;
```

```
for (j = 0; j < y_layer; j++) {
    cur_pos1 += bcols;
    tmpptr = cur_pos1;
    R2 = *tmpptr;
    rowptr = arow;
    R0 = *rowptr;
    /* broadcast multiplier across the columns */
    if (ixproc == x_index) {
     R2 *= scale;

     *cur_pos1 = R2;
     xnetcE[nxproc].mult = R2;
    }

    /* Update b values */

    *tmpptr3 = *tmpptr3 - mult*temp;
    tmpptr3++;


    /* subtract row from (multiplier * pivot_row) */
    for (k = 0; k < col_loops2; k++) {
     R0 *= mult;
     R1 = *++rowptr;
     R2 -= R0;
     *tmpptr++ = R2;

     R3 = *tmpptr;
     R0 = *++rowptr;
     R1 *= mult;
     R3 -= R1;
     *tmpptr++ = R3;
     R2 = *tmpptr;
    }

    R0 *= mult;
    R1 = *++rowptr;
    R2 -= R0;
    *tmpptr++ = R2;
    R3 = *tmpptr;
    R1 *= mult;
    R3 -= R1;
    *tmpptr = R3;
```

```
}

/* Updates the y_layer */
if (iyproc != y_index) { /* if off the row with pivot PE in y_layer */

    /* broadcast multipliers across the columns */
    tmpptr = cur_pos;
    R2 = *tmpptr;
    rowptr = arow;
    R0 = *rowptr;
    if (ixproc == x_index) {
     R2 *= scale;

     *cur_pos = R2;
     xnetcE[nxproc].mult = R2;
    }

    /* Update b values */

    *tmpptr3 = *tmpptr3 - mult*temp;

    /* subtract row from (multiplier * pivot_row) */
    for (k = 0; k < col_loops2; k++) {
     R0 *= mult;
     R1 = *++rowptr;
     R2 -= R0;
     *tmpptr++ = R2;

     R3 = *tmpptr;
     R0 = *++rowptr;
     R1 *= mult;
     R3 -= R1;
     *tmpptr++ = R3;
     R2 = *tmpptr;
    }
    R0 *= mult;
    R1 = *++rowptr;
    R2 -= R0;
    *tmpptr++ = R2;
    R3 = *tmpptr;
    R1 *= mult;
    R3 -= R1;
    *tmpptr = R3;
```

```
}

*tmpptr3++;

/* Updates rows in the layers below the y_layer (current layer) */
cur_pos1 = cur_pos;
for (j = y_layer+1; j < brows; j++) {
    cur_pos1 += bcols;
    tmpptr = cur_pos1;
    R2 = *tmpptr;
    rowptr = arow;
    R0 = *rowptr;

    if (ixproc == x_index) {
     R2 *= scale;

     *cur_pos1 = R2;
     xnetcE[nxproc].mult = R2;
    }

    /* Update b values */
    *tmpptr3 = *tmpptr3 - mult*temp;
    tmpptr3++;
    /* subtract row from (multiplier * pivot_row) */
    for (k = 0; k < col_loops2; k++) {
     R0 *= mult;
     R1 = *++rowptr;
     R2 -= R0;
     *tmpptr++ = R2;

     R3 = *tmpptr;
     R0 = *++rowptr;
     R1 *= mult;
     R3 -= R1;
     *tmpptr++ = R3;
     R2 = *tmpptr;
    }

    R0 *= mult;
    R1 = *++rowptr;
    R2 -= R0;
    *tmpptr++ = R2;
    R3 = *tmpptr;
```

```
        R1 *= mult;
        R3 -= R1;
        *tmpptr = R3;
      }
    }

    else {  /********** else if the # of remaining column layers is even then ***********/

     /* find maximal pivot - divide and conquer */

     /* find the maximal pivot on each PE in the column first */
     maxval = -1.0;
     if (ixproc == x_index) {
        /* tmpptr set to ptr. to the last elt. in column */
        tmpptr = cur_pos + (brows-y_layer-1)*bcols;

        /* software pipelined */
        R0 = *tmpptr;
        for (j = brows-1; j > y_layer; j--) {
          tmpptr -= bcols;
          R1 = *tmpptr;

          if (R0 < 0) R0 = -R0;
          if (R0 > maxval) {
            maxval = R0;
            maxloc = j;
          }
          R0 = R1;
        }
        if (R0 < 0) R0 = -R0;
        if (iyproc >= y_index) {
          if (R0 > maxval) {
            maxval = R0;
            maxloc = j;
          }
        }

        Loc1 = iyproc;
        /* find the maximal pivot on all PEs in the pivot column */
        max_to_row_0(maxval,R0,Loc1,Loc2);
        /* loc is the iyproc of max. elt */
        loc = proc[0][x_index].Loc1;
/* added to get average exchange distance with zeros in memory */
```

```
loc = (y_index + nyproc/2) % nyproc;

   }


    /* maxpost is y_layer in which the max. pivot elt found */
       maxpost = proc[loc][x_index].maxloc;



    /* maxptr is set to the last elt in row where pivot elt found */
    maxptr = curptr + maxpost*bcols;
    /* tmpptr2 ptrs to last elt in arow so a copy of the pivot row can be
         stored starting at the front of arow */
    tmpptr2 = arow + bcols - 1 - x_layer;


    /* try to swap the b values of the swapped rows */
    /* tmpptr3 ptrs to the elt in the b vector corresponding to the row where
         the max. pivot elt was found */
    tmpptr3 = b + maxpost;
    /* tmpptr4 ptrs to the elt in the b vector corresponding to row i */
    tmpptr4 = b + y_layer;

    temp = proc[loc][loc].*tmpptr3;
    proc[loc][loc].*tmpptr3 = proc[y_index][y_index].*tmpptr4;
    proc[y_index][y_index].*tmpptr4 = temp;

/*
    printf("temp = %f, loc = %d, maxdist = %d, maxpost = %d\n",temp,loc,maxdist,
          maxpost);
*/
    tmpptr = a;


    /*
     * broadcast the pivot row below to the rows
     * all values of arow that represent ELEMS to the left
     * of the current pivot remain 0.0
     *
     * interchange rows
     */
    if (iyproc == loc) { /* if PE in the row containing the max. pivot elt */
```

```
if (maxdist = (loc - y_index)) { /* if rows are on different PEs */
 /* maxdist is the distance in rows for which the swap must
    be performed  */
 /* tmpptr set to pt to the last elt in ith row */
 tmpptr = curptr + y_layer*bcols;
 /* PR0 pts to the last elt in the row of the max. pivot elt */
 PR0 = maxptr;

 /* Determine the shortest direction for swapping the rows */
 if (maxdist > half_nyproc) maxdist -= nyproc;
 else if (maxdist < -half_nyproc) maxdist += nyproc;

 if (maxdist < 0) { /* if ith row PE "above" where max. pivot found */
  maxdist = -maxdist;

  /* swap rows: software pipelined and loop unrolled to a
     depth of 2 */
  R0 = *maxptr;
  for (j = 0; j < col_loops2; j++) {
   R1 = *--PR0;
   xnetcS[nyproc].R2 = R0;
   all *tmpptr2-- = R2;
   R4 = xnetpS[maxdist].*tmpptr;
   *maxptr = R4;
   all if (iyproc == y_index) *tmpptr-- = R2;
   maxptr = PR0;
   R3 = *--PR0;
   xnetcS[nyproc].R2 = R1;
   all *tmpptr2-- = R2;
   R4 = xnetpS[maxdist].*tmpptr;
   *maxptr = R4;
   all if (iyproc == y_index) *tmpptr-- = R2;
   maxptr = PR0;
   R0 = R3;
  }

  R4 = xnetpS[maxdist].*tmpptr;
  *maxptr = R4;
  xnetcS[nyproc].R2 = R0;
  maxdist = -maxdist;
 }
 else { /* if ith row "below" where the max. pivot row found */
  R0 = *maxptr;
```

```
    for (j = 0; j < col_loops2; j++) {
      R1 = *--PR0;
      xnetcS[nyproc].R2 = R0;
      all *tmpptr2-- = R2;
      R4 = xnetpN[maxdist].*tmpptr;
      *maxptr = R4;
      all if (iyproc == y_index) *tmpptr-- = R2;
      maxptr = PR0;

      R3 = *--PR0;
      xnetcS[nyproc].R2 = R1;
      all *tmpptr2-- = R2;
      R4 = xnetpN[maxdist].*tmpptr;
      *maxptr = R4;
      all if (iyproc == y_index) *tmpptr-- = R2;
      maxptr = PR0;
      R0 = R3;
     }
    R4 = xnetpN[maxdist].*tmpptr;
    *maxptr = R4;
    xnetcS[nyproc].R2 = R0;
   }
  }
  else { /* else the rows to be swapped are on the same PEs */
   tmpptr = curptr + y_layer*bcols;
   PR0 = maxptr;

   if (tmpptr == maxptr) { /* if no swapped of rows necessary then
                              only broadcast pivot row */
    R0 = *maxptr;
    for (j = 0; j < col_loops2; j++) {
      R1 = *--PR0;
      xnetcS[nyproc].R2 = R0;
      all *tmpptr2-- = R2;
      *tmpptr-- = R2;
      maxptr = PR0;

      R3 = *--PR0;
      xnetcS[nyproc].R2 = R1;
      all *tmpptr2-- = R2;
      *tmpptr-- = R2;
      maxptr = PR0;
      R0 = R3;
```

```
        }
          xnetcS[nyproc].R2 = R0;
        }
        else {   /* swap rows on the same PEs and broadcast pivot row */


         R0 = *maxptr;
         for (j = 0; j < col_loops2; j++) {
          R1 = *--PR0;
          xnetcS[nyproc].R2 = R0;
          all *tmpptr2-- = R2;
          *maxptr = *tmpptr;
          *tmpptr-- = R2;
          maxptr = PR0;

          R3 = *--PR0;
          xnetcS[nyproc].R2 = R1;
          all *tmpptr2-- = R2;
          *maxptr = *tmpptr;
          *tmpptr-- = R2;
          maxptr = PR0;
          R0 = R3;
         }
          *maxptr = *tmpptr;
          xnetcS[nyproc].R2 = R0;
        }
      }
}

if (ixproc > x_index) {
    *tmpptr2 = R2;
}
else {
    *tmpptr2 = 0.0;
}

if (iyproc == y_index) {
    *tmpptr-- = R2;
}

if (ixproc == x_index)
    scale = 1.0 / proc[y_index][x_index].R2;
```

```
/*
 * for each subrow
 */

tmpptr3 = b;

/* Updates rows in the layers above the y_layer (current layer) */
cur_pos1 = cur_pos - y_layer*bcols - bcols;

for (j = 0; j < y_layer; j++) {
    cur_pos1 += bcols;
    tmpptr = cur_pos1;
    R2 = *tmpptr;
    rowptr = arow;
    R0 = *rowptr;
    /* broadcast multiplier across the columns */
    if (ixproc == x_index) {
     R2 *= scale;

     *cur_pos1 = R2;
     xnetcE[nxproc].mult = R2;
    }

    /* Update b values */

    *tmpptr3 = *tmpptr3 - mult*temp;
    tmpptr3++;


    /* subtract row from (multiplier * pivot_row) */
    for (k = 0; k < col_loops2; k++) {
     R0 *= mult;
     R1 = *++rowptr;
     R2 -= R0;
     *tmpptr++ = R2;

     R3 = *tmpptr;
     R0 = *++rowptr;
     R1 *= mult;
     R3 -= R1;
     *tmpptr++ = R3;
     R2 = *tmpptr;
    }
```

```
    R0 *= mult;
    R2 -= R0;
    *tmpptr = R2;
}

/* Updates the y_layer */
if (iyproc != y_index) { /* if off the row with pivot PE in y_layer */

    /* broadcast multipliers across the columns */
    tmpptr = cur_pos;
    R2 = *tmpptr;
    rowptr = arow;
    R0 = *rowptr;
    if (ixproc == x_index) {
     R2 *= scale;

     *cur_pos = R2;
     xnetcE[nxproc].mult = R2;
    }

    /* Update b values */

    *tmpptr3 = *tmpptr3 - mult*temp;

    /* subtract row from (multiplier * pivot_row) */
    for (k = 0; k < col_loops2; k++) {
     R0 *= mult;
     R1 = *++rowptr;
     R2 -= R0;
     *tmpptr++ = R2;

     R3 = *tmpptr;
     R0 = *++rowptr;
     R1 *= mult;
     R3 -= R1;
     *tmpptr++ = R3;
     R2 = *tmpptr;
    }
    R0 *= mult;
    R2 -= R0;
    *tmpptr = R2;
}
```

```
tmpptr3++;

/* Updates rows in the layers below the y_layer (current layer) */
cur_pos1 = cur_pos;
for (j = y_layer+1; j < brows; j++) {
    cur_pos1 += bcols;
    tmpptr = cur_pos1;
    R2 = *tmpptr;
    rowptr = arow;
    R0 = *rowptr;
    /* broadcast multiplier across the columns */
    if (ixproc == x_index) {
     R2 *= scale;

     *cur_pos1 = R2;
     xnetcE[nxproc].mult = R2;
    }

    /* Update b values */

    *tmpptr3 = *tmpptr3 - mult*temp;
    tmpptr3++;

    /* subtract row from (multiplier * pivot_row) */
    for (k = 0; k < col_loops2; k++) {
     R0 *= mult;
     R1 = *++rowptr;
     R2 -= R0;
     *tmpptr++ = R2;

     R3 = *tmpptr;
     R0 = *++rowptr;
     R1 *= mult;
     R3 -= R1;
     *tmpptr++ = R3;
     R2 = *tmpptr;
    }
     R0 *= mult;
     R2 -= R0;
     *tmpptr = R2;
 }
}
```

```
  /*
   * save reciprocals of diagonal elements
   */
  if (ixproc == x_index && iyproc == y_index) {
    *cur_pos = scale;
  }
 }  /* end for i = ... */

 x_layer = i>>lnxproc;
 y_layer = i>>lnyproc;
 x_index = i - (x_layer<<lnxproc);
 y_index = i - (y_layer<<lnyproc);

 cur_pos = a + y_layer*bcols + x_layer;

/* Divide b's by diagonal elements */
 tmpptr = a;
 R0 = *tmpptr;
 tmpptr += bcols + 1;
 tmpptr2 = b;
 R1 = *tmpptr2;
 for (i = 0; i < brows-1; i++) {
  R3 = *tmpptr;
  tmpptr += bcols + 1;
  R4 = *(tmpptr2+1);

  *tmpptr2++ = R1*R0;


  R0 = R3;
  R1 = R4;
 }


 if (ixproc == iyproc) {
  xnetcE[nxproc].R2 = R0;
 }

 i = n-bcols*nyproc;
 if (i <= 0)
  i += nyproc;
/* The last layer of diagonal elements contain the diagonal elts and not
```

```
their recipocals (scale).  */
 if (iyproc < i) {

   *tmpptr2 = R1*R0;

 }

p_free(arow);
}
```

B.3. Fast Fourier Transform Code

The FFT mpl code is divided into several files. The files and a brief description of what each file contains is as follows:

makefile - contains a make file to compile the FFT code

fft.h - contains the constant declarations for the size of the arrays used in the program (This must be changed if larger memory per processor is available)

fft.m - contains the main function that generates the data for the FFT, calls the fft function, and prints the results

fftio.m - contains the functions to generate test data, read and write data from the PE array

fftutil.m - contains the function that performs the fft as well as macros to perform complex arithmetic

```
#####################################################################
#####################################################################
.SUFFIXES: .o .m .c
MPFLAGS = -Zq -Zn -nohprofile -Omax

all: fft

fft: fft.h fftutil.o fftio.o fft.m
	mpl_cc $(MPFLAGS) fftio.o fftutil.o -o fft fft.m;mplimit fft pmem 16k

fft2: fft.mp2.h fftutil.o fftio2.o fft2.m
	mpl_cc $(MPFLAGS) fftio2.o fftutil.o -o fft2 fft2.m;mplimit fft pmem 64k

sfft: fft.h fftutil.S fftio.o fft.m
	mpl_cc $(MPFLAGS) fftio.o fftutil.S -o sfft fft.m;mplimit fft pmem 16k

sfft2: fft.mp2.h fftutil.S fftio2.o fft2.m
	mpl_cc $(MPFLAGS) fftio2.o fftutil.S -o sfft2 fft2.m;mplimit fft pmem 64k

fftio.o: fft.h fftio.m
	mpl_cc $(MPFLAGS) -c fftio.m

fftio2.o: fft.mp2.h fftio2.m
	mpl_cc $(MPFLAGS) -c fftio2.m

fftutil.o: fft.h fftutil.m
	mpl_cc $(MPFLAGS) -c fftutil.m
```

```
/*-------------------------------------------------------------------
File: fft.h
Programmer: Mark Fienup
Description: Declaration of the maximum matrices' sizes
-------------------------------------------------------------------*/
#include <mpl.h>
#include <stdio.h>
#include <math.h>

#define MAXLEN 1024
#define MAXBUF 1024
#define HALFMAXBUF 512
```

```
/*-----------------------------------------------------------------
File: fft.m
Programmer: Mark Fienup
Description: Main program for the binary exchange FFT
(MasPar mpl code)
Usage: fft number, where number is the log2(N)
-----------------------------------------------------------------*/

#include <stdio.h>
#include <math.h>
#include "fft.h"

/* Storage for the real and imaginary parts of the data elements */
plural float a_real[MAXBUF];
plural float a_imag[MAXBUF];

/*  Storage for twiddles */
plural float wm_real[HALFMAXBUF];
plural float wm_imag[HALFMAXBUF];

/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
main(argc, argv)
  int argc;
  char *argv[];
  {
  unsigned rows, cols;
  unsigned logrows;
  unsigned n, s, m, j, k;
  int powerOfTwo;

  if (argc != 2) {
    fprintf( stderr, "Usage:  %s number\n", argv[0]);
    exit(0);
  }

/* log2 of the number of data points */
  powerOfTwo = atoi(argv[1]);
  n = power2(powerOfTwo);

/* Generate the data points */
  fft_gendata(a_real, a_imag, n);
```

```
/* Perform the FFT */
  fft(a_real, a_imag, wm_real, wm_imag, n);

/* Save the results to disk */
  fft_bwtc(stdout, a_real, a_imag, n);

}
```

```
/*-------------------------------------------------------------------
File: fftio.m
Programmer: Mark Fienup
Description: I/O functions for FFT
(MasPar mpl code)
Usage: fft number, where number is the log2(N)
-------------------------------------------------------------------*/

/*********************************************************************/
/*********************************************************************/
#include <mpl.h>
#include <stdio.h>
#include "fft.h"


/*-------------------------------------------------------------------*/
/* Function: fft_gendata()    (fft generate complex test data*/
/*-------------------------------------------------------------------*/
fft_gendata (m_real, m_imag, nelems)
  plural float   m_real[], m_imag[];
  unsigned       nelems;
  {
  float  tmp_real, tmp_imag;
  unsigned pebits, pemask, pe;
  unsigned halfelems;
  unsigned i;


  if (nelems/nproc > MAXBUF)
    {
    fprintf(stderr, "fft_gendata: Matrix too large\n");
    return -1;
    }

  pebits   = log2(nproc);
  pemask   = nproc-1;
  halfelems = nelems>>1;
  tmp_imag = 0.0;

  for (i=0; i<nelems; i++) {
    pe = i&pemask;
    proc[pe].(m_real[(i>>pebits)]) = (float) i;
    proc[pe].(m_imag[(i>>pebits)]) = tmp_imag;
```

```
    }
  return 0;
  }

/*-------------------------------------------------------------------*/
/* Function: fft_bwtc()    (FFT binary write complex)        */
/*-------------------------------------------------------------------*/
fft_bwtc(fp, m_real, m_imag, nelems)
  FILE          *fp;
  plural float   m_real[], m_imag[];
  unsigned       nelems;
  {
  float  tmp_real, tmp_imag;
  unsigned bufsize, halfelems, logelems;
  unsigned pebits, pemask;
  unsigned revbits, pe, offset, offsetbits, offsetmask;
  unsigned i, cols;

  cols = 2;

  if (fwrite(&nelems, sizeof(nelems), 1, fp)!=1)
    {
    fprintf(stderr, "fft_bwtc: error writing rows\n");
    return -1;
    }
  if (fwrite(&cols, sizeof(cols), 1, fp)!=1)
    {
    fprintf(stderr, "fft_bwtc: error writing cols\n");
    return -1;
    }

  if (nelems/nproc > MAXBUF)
    {
    fprintf(stderr, "fft_bwtc: Matrix dimensions too large\n");
    return -1;
    }

  logelems  = log2(nelems);
  for (i=0; i<nelems; i++)
    {
    revbits = bitrev(i,logelems);
    pe      = (revbits / 2) % nproc;
```

```
  offset  = (((revbits / 2) / nproc) * 2) + (revbits & 01);

  tmp_real= proc[pe].(m_real[offset]);
  tmp_imag = proc[pe].(m_imag[offset]);
  if (fwrite(&tmp_real, sizeof(tmp_real), 1, fp) != 1)
    {
    fprintf(stderr, "fft_brdc: Error writing matrix element %u\n",i);
    return -1;
    }
  if (fwrite(&tmp_imag, sizeof(tmp_imag), 1, fp) != 1)
    {
    fprintf(stderr, "fft_brdc: Error writing matrix element %u\n",i);
    return -1;
    }
  }
return 0;
}
```

```
/*-------------------------------------------------------------------
File: fftutil.m
Programmer: Mark Fienup
Description: FFT function and macros to perform complex
arithmetic
-------------------------------------------------------------------*/

#include <mpl.h>
#include <stdio.h>
#include <math.h>
#include "/usr/maspar/include/ampl/maspar/values.h"

void dpuTimerStart();
double dpuTimerElapsed();

/*-------------------------------------------------------------------*/
/* Reused as many register variables as possible                     */
/*-------------------------------------------------------------------*/
#define twiddle_real a_real_base_offset2
#define twiddle_imag a_imag_base_offset2
#define down_real a_real_base2
#define down_imag a_imag_base2
#define xdist dist
#define ydist dist
#define base active

/*-------------------------------------------------------------------*/
/* Complex number macros                                             */
/*-------------------------------------------------------------------*/
#define cneg(c_real, c1_real, c_imag, c1_imag)   c_real = - c1_real;\
  c_imag = - c1_imag

/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
#define cadd(c_real, c1_real, c2_real, c_imag, c1_imag, c2_imag) \
  c_real = c1_real + c2_real;\
  c_imag = c1_imag + c2_imag

/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
#define caddp(c, c1, c2)   c->real = c1_real + c2_real;\
  c->imag = c1_imag + c2_imag
```

```
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
#define csub(c_real, c1_real, c2_real, c_imag, c1_imag, c2_imag) \
  c_real = c1_real - c2_real;\
  c_imag = c1_imag - c2_imag

/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
#define cmult(c_real, c1_real, c2_real, c_imag, c1_imag, c2_imag) \
  multtemp = (c1_real * c2_real) - (c1_imag * c2_imag);\
  c_imag = (c1_real * c2_imag) + (c1_imag * c2_real);\
  c_real = multtemp

/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
#define cmultp(c, c1, c2) \
  multtemp_real = (c1_real * c2_real) - (c1_imag * c2_imag);\
  c->imag = (c1_real * c2_imag) + (c1_imag * c2_real);\
  c->real = multtemp_real

/*-------------------------------------------------------------------*/
/*  Utility functions                                                */
/*-------------------------------------------------------------------*/
unsigned bitrev(bits,loglen)
  unsigned bits, loglen;
  {
  unsigned rbits;
  rbits = 0;
  while (loglen--)
    {
    rbits <<= 1;
    rbits |= (bits&01);
    bits >>= 1;
    }
  return rbits;
  }

/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
unsigned power2(x)
  unsigned x;
  {
  unsigned result;
```

```
  for (result=1; x; result<<=1,x--);
  return result;
  }

/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
unsigned log2(num)
  unsigned num;
  {
  unsigned i,log;
  switch (num)
    {
    case 1:  return 0;
    case 2:  return 1;
    default: for (log=1, i=(num>>2); i; i>>=1)
               log++;
    }
  return log;
  }


/*-----------------------------------------------------------------*/
/*  The FFT functions (square-of-twiddles)                         */
/*-----------------------------------------------------------------*/

fft(a_real, a_imag, wm_real, wm_imag, n)
  plural float a_real[], a_imag[];
  plural float wm_real[], wm_imag[];
  unsigned n;

{
  /* Define register variables */
  register plural float up_real, temp_real /*, down_real, twiddle_real*/;
  register plural float up_imag, temp_imag /*, down_imag, twiddle_imag*/;
  register plural float a_real_base, a_imag_base;
  register plural float a_real_base2, a_imag_base2;
  register plural float a_real_base_offset, a_imag_base_offset;
  register plural float a_real_base_offset2, a_imag_base_offset2;

  register plural float multtemp;
  register plural float *pa_real;
  register plural float *pa_imag;
  register plural float *pwm_real;
```

```
  register plural float *pwm_imag, pi;

  register unsigned bufsize, dist, offset, base, current_base;
  register unsigned butterfly, butterflies, /* active,*/ lowerhalf;
  register plural float dtemp, dnproc, dn, dpower;
  double calcTime, totalTime, twiddleTime;

  dpuTimerStart();
  twiddleTime = dpuTimerElapsed();

  dpuTimerStart();

/* Calculate the local twiddles needed initially */
  pi = (plural float) M_PI;
  pi = pi*2;

  bufsize = n / nproc;
  butterflies = bufsize >> 1;
  dpower = (plural float) iproc;
  dn  = (plural float) n;
  dnproc = (plural float) nproc;


/* Calculate initial twiddle factors */

  pwm_real = &wm_real[0];
  pwm_imag = &wm_imag[0];
  for (butterfly=0; butterfly < butterflies; butterfly++) {
    dtemp  = (pi * dpower)/dn;

    *pwm_real++ = fp_cos(dtemp);
    *pwm_imag++ = fp_sin(dtemp);

    dpower = dpower + dnproc;
   }

  twiddleTime = dpuTimerElapsed();

/* In memory stages of the fft */
/*  Software pipelined */

  for (offset=bufsize>>1; offset > 1; offset>>=1 ) {
    base = 0;
```

```
lowerhalf = offset >> 1;
pwm_real = &wm_real[0];
pwm_imag = &wm_imag[0];

a_real_base = a_real[base];
a_imag_base = a_imag[base];

a_real_base_offset = a_real[base+offset];
a_imag_base_offset = a_imag[base+offset];

twiddle_real = *pwm_real;
twiddle_imag = *pwm_imag;

for (butterfly=0; butterfly<butterflies; butterfly++) {
  current_base = base;
  base = base + 1;
  if ( (offset&base) != 0 ) {
    base = base + offset;
   }

  /* Perform butterfly operation */
  cadd(up_real, a_real_base, a_real_base_offset,
     up_imag, a_imag_base, a_imag_base_offset);
  csub(temp_real, a_real_base, a_real_base_offset,
     temp_imag, a_imag_base, a_imag_base_offset);

  a_real_base = a_real[base];
  a_imag_base = a_imag[base];

  a_real_base_offset = a_real[base+offset];
  a_imag_base_offset = a_imag[base+offset];

  cmult(a_real[current_base+offset], temp_real, twiddle_real,
      a_imag[current_base+offset], temp_imag, twiddle_imag);
  a_real[current_base] = up_real;
  a_imag[current_base] = up_imag;

  /* Update the twiddle factors by squaring it */
  cmult(twiddle_real, twiddle_real, twiddle_real,
      twiddle_imag, twiddle_imag, twiddle_imag);

  if ( (butterfly&lowerhalf) != 0 ) {
    cneg(twiddle_real, twiddle_real,
```

```
        twiddle_imag, twiddle_imag);
    }
    *pwm_real = twiddle_real;
    *pwm_imag = twiddle_imag;
    *pwm_real++;
    *pwm_imag++;

    twiddle_real = *pwm_real;
    twiddle_imag = *pwm_imag;


  }
}


  base = 0;
  offset = 1;
  pwm_real = &wm_real[0];
  pwm_imag = &wm_imag[0];

  a_real_base = a_real[base];
  a_imag_base = a_imag[base];

  a_real_base_offset = a_real[base+offset];
  a_imag_base_offset = a_imag[base+offset];

  twiddle_real = *pwm_real;
  twiddle_imag = *pwm_imag;

  for (butterfly=0; butterfly<butterflies; butterfly++) {
    current_base = base;
    base = base + 1;
    if ( (offset&base) != 0 ) {
      base = base + offset;
    }

    cadd(up_real, a_real_base, a_real_base_offset,
       up_imag, a_imag_base, a_imag_base_offset);
    csub(temp_real, a_real_base, a_real_base_offset,
       temp_imag, a_imag_base, a_imag_base_offset);

    a_real_base = a_real[base];
    a_imag_base = a_imag[base];
```

```
    a_real_base_offset = a_real[base+offset];
    a_imag_base_offset = a_imag[base+offset];

    cmult(a_real[current_base+offset], temp_real, twiddle_real,
        a_imag[current_base+offset], temp_imag, twiddle_imag);
    a_real[current_base] = up_real;
    a_imag[current_base] = up_imag;

    cmult(twiddle_real, twiddle_real, twiddle_real,
        twiddle_imag, twiddle_imag, twiddle_imag);

    if ( iproc >= (nproc>>1) ) {
      cneg(twiddle_real, twiddle_real,
          twiddle_imag, twiddle_imag);
    }
    *pwm_real = twiddle_real;
    *pwm_imag = twiddle_imag;
    *pwm_real++;
    *pwm_imag++;

    twiddle_real = *pwm_real;
    twiddle_imag = *pwm_imag;

  }


/* Prepare to perform communication stages of FFT */
/* Load registers before first stage and save at the end */
pa_real = &a_real[0];
pa_imag = &a_imag[0];
pwm_real = &wm_real[0];
pwm_imag = &wm_imag[0];
for (butterfly=0; butterfly<butterflies; butterfly++) {
  up_real = *pa_real++;
  up_imag = *pa_imag++;
  down_real = *pa_real;
  down_imag = *pa_imag;

  twiddle_real = *pwm_real++;
  twiddle_imag = *pwm_imag++;

  /* Stages requiring communication along the y dimension of the PE array */
```

```
active = nproc >> 1;
for (ydist=nyproc>>1; ydist > 0; ydist >>=1) {
  temp_real = down_real;
  temp_imag = down_imag;

  if ((active&iproc) == 0) {
    down_real = xnetS[ydist].up_real;
    down_imag = xnetS[ydist].up_imag;
  } else {
    up_real = xnetN[ydist].temp_real;
    up_imag = xnetN[ydist].temp_imag;
  }

  temp_real = up_real;
  temp_imag = up_imag;

  cadd(up_real, up_real, down_real,
     up_imag, up_imag, down_imag);
  csub(down_real, temp_real, down_real,
     down_imag, temp_imag, down_imag);
  cmult(down_real, down_real, twiddle_real,
       down_imag, down_imag, twiddle_imag);

  cmult(twiddle_real, twiddle_real, twiddle_real,
      twiddle_imag, twiddle_imag, twiddle_imag);

  if (((active>>1)&iproc) != 0 ) {
    cneg(twiddle_real, twiddle_real,
        twiddle_imag, twiddle_imag);
  }

  active >>=1;
}


/* Stages requiring communication along the X dimension of the PE array */

for (xdist=nxproc>>1; xdist > 0; xdist >>=1) {

  temp_real = down_real;
  temp_imag = down_imag;
```

```
    if ((active&iproc) == 0) {
      down_real = xnetE[xdist].up_real;
      down_imag = xnetE[xdist].up_imag;
    } else {
      up_real = xnetW[xdist].temp_real;
      up_imag = xnetW[xdist].temp_imag;
    }

    temp_real = up_real;
    temp_imag = up_imag;

    cadd(up_real, up_real, down_real,
        up_imag, up_imag, down_imag);
    csub(down_real, temp_real, down_real,
        down_imag, temp_imag, down_imag);
    cmult(down_real, down_real, twiddle_real,
        down_imag, down_imag, twiddle_imag);

    cmult(twiddle_real, twiddle_real, twiddle_real,
        twiddle_imag, twiddle_imag, twiddle_imag);

    if (((active>>1)&iproc) != 0 ) {
      cneg(twiddle_real, twiddle_real,
          twiddle_imag, twiddle_imag);
    }

  active >>=1;
  }

  /* Store the results */
  *(pa_real-1) = up_real;
  *(pa_imag-1) = up_imag;
  *pa_real = down_real;
  *pa_imag = down_imag;

  pa_real++;
  pa_imag++;


}

/* Print timing information */
totalTime = dpuTimerElapsed();
```

```
  calcTime = totalTime - twiddleTime;

  fprintf(stderr, "total time = %lf  ", totalTime);
  fprintf(stderr, "Twiddle  Time = %lf  (%lf percent)  ", twiddleTime,
          ((100.0 * twiddleTime)/totalTime));
  fprintf(stderr, "FFT Calc. Time = %lf  (%lf percent)\n", calcTime,
          ((100.0 * calcTime)/totalTime));

}
```